

# **uaDB Developer's Guide**

M. C. Witthoeft<sup>1</sup>, J. González<sup>2</sup>, T. K. Kallman<sup>1</sup>, J. García<sup>1,3</sup>

<sup>1</sup> NASA Goddard Space Flight Center, Greenbelt, MD, USA

<sup>2</sup> Instituto Venezolano de Investigaciones Científicas (IVIC),  
Caracas Venezuela

<sup>3</sup> Catholic University of America, Washington D.C., USA

December 23, 2011



# Contents

<b>Contents</b>	<b>3</b>
<b>List of Tables</b>	<b>7</b>
<b>List of Figures</b>	<b>9</b>
<b>1 Introduction</b>	<b>11</b>
<b>2 Database Structure</b>	<b>13</b>
2.1 Description . . . . .	13
2.2 Store2/3 tables . . . . .	14
2.3 Table definitions . . . . .	15
2.3.1 <i>elements</i> . . . . .	16
2.3.2 <i>ions</i> . . . . .	16
2.3.3 <i>allconfs</i> . . . . .	17
2.3.4 <i>allterms</i> . . . . .	17
2.3.5 <i>alllevels</i> . . . . .	17
2.3.6 <i>sources</i> . . . . .	18
2.3.7 <i>reference</i> . . . . .	18
2.3.8 <i>sourceref</i> . . . . .	18
2.3.9 <i>fundamental</i> . . . . .	19
2.3.10 <i>datagroups</i> . . . . .	19
2.3.11 <i>datatype</i> . . . . .	19
2.3.12 <i>paramtypes</i> . . . . .	20
2.3.13 <i>paramdef</i> . . . . .	20
2.3.14 <i>setdef</i> . . . . .	20
2.3.15 <i>paramgroups</i> . . . . .	21
2.3.16 <i>groupdef</i> . . . . .	21
2.3.17 <i>data</i> . . . . .	21
2.3.18 <i>door2</i> . . . . .	21
2.3.19 <i>store2</i> . . . . .	22
2.3.20 <i>door3</i> . . . . .	22
2.3.21 <i>store3</i> . . . . .	22
2.3.22 <i>parameters</i> . . . . .	23

2.3.23 <i>units</i> . . . . .	23
2.3.24 <i>unitalias</i> . . . . .	23
2.3.25 <i>unitconvert</i> . . . . .	24
2.3.26 <i>notes</i> . . . . .	24
2.3.27 <i>log</i> . . . . .	24
<b>3 PHP Libraries</b>	<b>25</b>
3.1 <i>admin.library</i> . . . . .	25
3.1.1 Constants . . . . .	25
3.1.2 Fundamental data type categories . . . . .	26
3.1.3 Grid . . . . .	26
3.1.4 Data type categories . . . . .	26
3.1.5 Parameter group categories . . . . .	26
3.1.6 Parameter type categories . . . . .	26
3.1.7 System-related functions . . . . .	26
3.1.8 Conversion functions . . . . .	26
3.1.9 Physics related functions . . . . .	27
3.1.10 Translation functions . . . . .	27
3.1.11 General database functions . . . . .	27
3.1.12 Web page functions . . . . .	27
3.2 Table classes . . . . .	27
3.2.1 <i>adminTable</i> . . . . .	28
3.2.2 <i>adminData</i> . . . . .	28
3.2.3 <i>adminIons</i> . . . . .	29
3.2.4 <i>adminErrorHandler</i> . . . . .	29
3.2.5 <i>adminFiles</i> . . . . .	29
3.3 Web classes . . . . .	30
<b>4 PHP command line scripts</b>	<b>31</b>
4.1 <i>xsc_createdb.php</i> . . . . .	31
4.2 <i>xsc_delete.php</i> . . . . .	31
4.3 <i>xsc_optimize.php</i> . . . . .	32
4.4 <i>xsc_browser.php</i> . . . . .	32
4.5 <i>xsc_ld*.php</i> . . . . .	32
4.6 <i>docgen_tabledesc.php</i> . . . . .	32
<b>5 XML files</b>	<b>35</b>
5.1 Fundamental data type definitions . . . . .	35
5.2 Parameter type definitions . . . . .	36
5.3 Data type definitions . . . . .	36
5.4 xstar source file . . . . .	37
5.5 XML interface . . . . .	41
5.5.1 Examples for PARAM_CONNECT tags . . . . .	43
5.6 Python code reference . . . . .	44

<b>CONTENTS</b>	<b>5</b>
<b>6 Creating and populating UADB</b>	<b>47</b>
6.1 Creating the MySQL database . . . . .	47
6.2 Add definitions and types . . . . .	47
6.3 Adding xstar data . . . . .	48
6.3.1 Type 6 . . . . .	48
6.3.2 General insertion procedure . . . . .	49
6.3.3 type 50 . . . . .	49
6.3.4 type 66 . . . . .	50
6.3.5 type 71 . . . . .	50
<b>7 How-to Guide</b>	<b>51</b>
7.1 Adding special parameter types . . . . .	51
7.2 Processing PREP_DATA input files . . . . .	52
<b>A xstar data types</b>	<b>55</b>
A.1 Type 1 . . . . .	55
A.2 Type 2 . . . . .	56
A.3 Type 7 . . . . .	56
A.4 Type 10 . . . . .	57
A.5 Type 16 . . . . .	58
A.6 Type 25 . . . . .	59
A.7 Type 30 . . . . .	60
A.8 Type 38 . . . . .	60
A.9 Type 39 . . . . .	61
A.10 Type 49 . . . . .	62
A.11 Type 50 . . . . .	63
A.12 Type 51 . . . . .	63
A.13 Type 53 . . . . .	64
A.14 Type 54 . . . . .	65
A.15 Type 56 . . . . .	65
A.16 Type 57 . . . . .	66
A.17 Type 59 . . . . .	66
A.18 Type 60 . . . . .	67
A.19 Type 62 . . . . .	68
A.20 Type 63 . . . . .	69
A.21 Type 67 . . . . .	70
A.22 Type 68 . . . . .	70
A.23 Type 69 . . . . .	71
A.24 Type 70 . . . . .	72
A.25 Type 71 . . . . .	72
A.26 Type 72 . . . . .	73
A.27 Type 73 . . . . .	73
A.28 Type 74 . . . . .	74
A.29 Type 76 . . . . .	75
A.30 Type 77 . . . . .	75
A.31 Type 79 . . . . .	76

A.32 Type 86 . . . . .	77
A.33 Type 88 . . . . .	77
<b>B Database input files</b>	<b>79</b>
B.1 PREP_ELEMENTS . . . . .	79
B.2 PREPIONS . . . . .	80
B.3 PREP_ALLLEVELS . . . . .	80
B.4 PREP_UNIT . . . . .	80
B.5 PREP_FUNDAMENTAL . . . . .	80
B.6 PREP_PARAMTYPE . . . . .	81
B.7 PREP_DATATYPE . . . . .	81
B.8 PREP_REFERENCE . . . . .	82
B.9 PREP_SOURCE . . . . .	82
B.10 PREP_DATA . . . . .	83
<b>C xstar Ion Numbers</b>	<b>87</b>
<b>D Configuration Strings</b>	<b>89</b>

# List of Tables

2.1	Average row size (in bytes) and average percentage indexing cost of database tables which store data. Also given is the effective row size which is simply the product of the first two rows. . . . .	16
3.1	Standardized instance names for each class. . . . .	27
5.1	Descriptors for XML file describing xstar source files. . . . .	40
5.2	Supported values of SUB tag in interface XML file. . . . .	43
B.1	List of input types. Some value types have square brackets; the character in the brackets is used to represent a space in the data value. Periods (.) are the default value, but can be replaced by any other character; for example [*]. . . . .	84
C.1	xstar ion index for all neutral ions up to zinc ( $Z = 30$ ). . . . .	88



# List of Figures

2.1 Schematic of all database tables. . . . .	15
5.1 Schematic of Python classes dealing with database/xstar interface.	45



# Chapter 1

## Introduction

In this document, the details of uaDB's implementation is laid out. We first describe the structure of the database tables and how the tables inter-relate. This is followed by a description of the PHP codes used to access the database. This discussion includes a tutorial on the command-line scripts used to modify the database contents. We also detail the PHP code underlying the web page.

XML is used to initially define data types, sub-types, and parameter types. These XML files are then parsed to create input files for the database. We also have XML files to describe the format of the input files for the xstar modeling code and further XML files which serve to connect the data in the xstar files with the xstarDB data sub-types. A chapter is included here discussing the format of all these XML files.

A note: when the database structure was first designed, the data types were called fundamental data types and data sub-types were called data types. For the rest of this document, we will use the original labels.



# Chapter 2

# Database Structure

## 2.1 Description

The uaDB SQL database currently consists of 27 tables. A schematic of the tables can be seen in Figure 2.1. Only six of these tables, *data* and *parameters*, are responsible for holding data. Consider each data set to be a mathematical function with  $n + m$  parameters:  $f(x_1, \dots, x_n, y_1, \dots, y_m)$ . The  $y_i$  parameters represent those parameters which change rapidly compared to the  $x_i$  parameters. An example is tabulated cross sections against energy, where for a given ion and transition ( $x_i$  parameters), there can be thousands of energy points ( $y_i$ ). If  $m = 0$ , then  $f$  is stored in *data* and the  $x_i$  values are stored in *parameters*. For  $m = 1$ , the  $f$  and  $y_i$  values are stored in the *store2* table, the  $x_i$  values are still in *parameters*, and in the *data* table is stored the number of points in the *store2* table for that set of  $x_i$  parameters. Connecting the *data* and *store2* tables is the *door2* table which contains summary information about the data set. The  $m = 2$  case is the same as  $m = 1$  substituting *store3* and *door3* for *store2* and *door2*. Larger data sets ( $m > 2$ ) are not currently supported.

Three tables are used to store source information. The *reference* table contains individual journal articles or books with references to the journal web page and NASA ADS entry when possible. The *sources* table is used to describe the origin of one or more data sets and is associated with zero or more references through the *sourceref* table.

Three tables are needed to define a specific data format. The *datatype* table describes a data value and the *paramtypes* table describes various parameter values. The *paramdef* table stores the combination of parameter types belonging to each data type. Each data type belongs to a data group (*datagroups*) which in turn belongs to a fundamental data type (*fundamental*). Data types belonging to the same data group can be easily converted between them while the fundamental data type defines a specific process (e.g. energy level or photoionization). Similarly parameter types can be organized into groups with the *paramgroups* table. Unlike data types, parameter types can belong to more than

one group. The table, *groupdef*, stores the membership list of each parameter group.

Elements and their ions are considered as special parameters and stored in their own tables: *elements* and *ions*. Data having an ion as a parameter will point to the appropriate row in the *ions* table. Levels are another special parameter and require the use of three tables: *allconfs*, *allterms*, and *allelevels*. The first of these defines configurations for any number of electrons (currently only up to 30 electrons). The *allterms* table defines all possible *LS* terms for each configuration; *LS* terms in this table points to specific configurations so that the total number of terms for each configuration is correct. The *allelevels* table defines all possible  $2J$  values for each term in the *allterms* table. Again, these tables are connected so that the level count is correct. The three level tables allow for a common level framework to underlay all the atomic data. This makes level mapping between data sets trivial and also allows for data using different coupling schemes to be used together.

There are three tables for storing information about units. The *units* table is the list of supported units and *unitalias* gives a list of alternate spellings for each unit. The *unitconvert* table stores usable conversion formulae between pairs of units.

The *notes* table is used to store extraneous information about data. There are currently two types of notes: labels and remarks. A remark is just a comment about a individual piece of data or a whole data set. If, for example, a data value in the database corrects an error in the source publication, a remark would be used to note this fact. Only one label can be applied to any data which will be used when displaying the data instead of the internally generated label. An example of label usage is giving a level a *jj*-coupling label when the internal *LS* label is not accurate.

Finally, the *log* table keeps track of changes to the database. It will not store the new/old data, but simply log the source and datatype where the changes were made. The name of the modifier is also stored along with the date of the modification.

## 2.2 Store2/3 tables

Before continuing, we want to remark on the efficiency gained by using the *store2* table. In the original form of the database, there was no distinction between the  $n x_i$  and  $m y_j$  parameters (taking the functional definition of a data set in the previous section). If a data set had  $N$  points, then it would occupy  $N$  rows in the *data* table and  $N(n+m)$  rows in the *parameters* table. The total amount of space taken up is

$$\Phi_{\text{old}} = N\phi_{\text{data}} \nu_{\text{data}} + N(n+m) \phi_{\text{parameters}} \nu_{\text{parameters}} \quad (2.1)$$

where  $\phi_{\text{table}}$  is the row size of *table* and  $\nu_{\text{table}}$  is the overhead cost due to indexing of the table. The indexing is what allows for quick searching of the tables. Using the *store2* ( $m=1$ ) and *store3* ( $m=2$ ) tables, the data set only requires a single

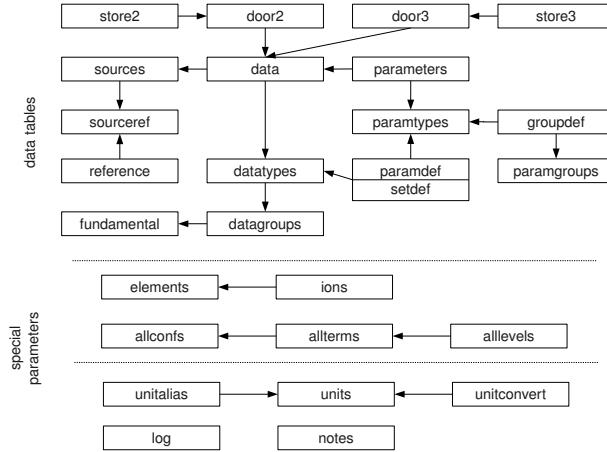


Figure 2.1: Schematic of all database tables.

row in the *data* table and  $n$  rows in the *parameters* table while the *store2/3* table requires  $N$  rows. The total size of the data is

$$\begin{aligned} \Phi_{\text{new}} = & \phi_{\text{data}} \nu_{\text{data}} + n \phi_{\text{parameters}} \nu_{\text{parameters}} \\ & + N \phi_{\text{store}} \nu_{\text{store}} \end{aligned} \quad (2.2)$$

where *store* refers to either *store2* or *store3* depending on the value of  $m$ . If  $N$  is large ( $\gtrsim 50$ ), the first two terms of the last expression can be ignored and the ratio of sizes becomes

$$\frac{\Phi_{\text{new}}}{\Phi_{\text{old}}} = \frac{\phi_{\text{store}} \nu_{\text{store}}}{\phi_{\text{data}} \nu_{\text{data}} + (n+m) \phi_{\text{parameters}} \nu_{\text{parameters}}}. \quad (2.3)$$

The row sizes, index costs, and effective row size (product of size and cost) for the relevant tables are given in Table 2.1. Transition data typically has  $n = 4$  which yields a ratio of  $\sim 19\%$  for both  $m = 2, 3$ , which means that the old method requires  $\sim 5$  times as much disk space as the new method using the *store2/3* tables. The savings comes not only from the large number of duplicate parameters being stored under the old method, but also from the improved indexing efficiency of the *store2/3* tables compared to the *parameters* table.

We should also note that, since the *store2* and *store3* tables are not accessed during a standard search, using these tables speeds up searching considerably.

## 2.3 Table definitions

The Universal Atomic Database consists of 27 interconnected tables. In this section, we will give the structure of each table and describe how the tables relate to each other.

table	size (B)	idx. cost (%)	eff. size (B)
data	32	206	66
parameters	30	227	68
store2	47	156	73
store3	64	139	89

Table 2.1: Average row size (in bytes) and average percentage indexing cost of database tables which store data. Also given is the effective row size which is simply the product of the first two rows.

### 2.3.1 *elements*

The *elements* table defines the set of atoms supported by xstarDB. Currently it includes all elements up to Zinc ( $Z = 30$ ). The table structure is

Field	Type	Description	Index
id	tinyint unsigned	primary key	primary
z	tinyint unsigned	nuclear charge	index
name	varchar(16)	full element name	
sym	varchar(4)	atomic symbol	
mass	varchar(32)	mass in amu	

Note: as the atomic mass is stored in this table, the database does not currently support isotopes.

### 2.3.2 *ions*

The *ions* table defines the full set of supported ions and is built on top of the *elements* table. The nuclear charge is defined as a pointer to the *elements* table. Despite being redundant, the number of electrons and the charge state are both specified to make it easy to search the database for either. The charge state,  $z_1$ , is defined according to astrophysical notation, that is  $z_1 = Z - n_e + 1$  where  $Z$  is the nuclear charge and  $n_e$  is the number of electrons. Finally, pointers to the ground configuration, term, and level are given. These pointers refer to the *allconfs*, *allterms*, and *alllevels* tables respectively.

Field	Type	Description	Index
id	smallint unsigned	primary key	primary
eid	tinyint unsigned	key from elements	index
ne	tinyint unsigned	number of electrons	index
charge	tinyint unsigned	charge state	index
gnd_conf	int unsigned	key from allconfs	
gnd_term	int unsigned	key from allterms	
gnd_level	int unsigned	key from alllevels	

### 2.3.3 *allconfs*

The *allconfs* table contains configurations for any number of electrons. The configuration is specified as a string described in Appendix D. To assist in searching, the number of electrons and parity are also stored for each configuration.

Field	Type	Description	Index
id	int unsigned	primary key	primary
conf	varchar(64)	configuration	index
ne	tinyint unsigned	number of electrons	index
parity	tinyint unsigned	configuration parity	

### 2.3.4 *allterms*

All possible *LS* terms belonging to the configurations in the *allconfs* table are listed in the *allterms* table. The configuration of each term is given by a pointer,cid, to the *allconfs* table. The multiplicity and angular momentum are given by *g* and *l*. Since duplicate (*g,l*) pairs can exist for a given configuration, an extra quantum number, alpha, is necessary to distinguish some terms. The alpha parameter is similar to but not the same as the seniority<sup>1</sup> of a term as the latter is not always unique. Instead, alpha is set to 1 for the first term having a given (*g,l*) pair and is incremented for each additional term having that pair. In practice, for a given term, alpha increases with energy.

Field	Type	Description	Index
id	int unsigned	primary key	primary
cid	int unsigned	key from allconfs	index
g	tinyint unsigned	2S+1	
l	tinyint unsigned	L	
alpha	tinyint unsigned	extra quantum number	
ne	tinyint unsigned	number of electrons	index

### 2.3.5 *alllevels*

Finally, all levels associated with the terms in *allterms* are given in the *alllevels* table. This simple table contains a pointer to the *allterms* table, tid, and twice the total angular momentum of the level, j2. The number of electrons is also stored to aid searching.

Field	Type	Description	Index
id	int unsigned	primary key	primary
tid	int unsigned	key from allterms	index
j2	tinyint unsigned	2J	
ne	tinyint unsigned	number of electrons	index

---

<sup>1</sup> R. D. Cowan, *The Theory of Atomic Structure and Spectra*, p. 261 (University of California Press, Berkeley, 1981)

### 2.3.6 *sources*

All data entered into the database must have a source. The *sources* table holds information for each source and the date when it is added. Scientific details of the source are stored in description, while other information possibly regarding how the data was inserted into the database is stored in notes. Finally, each source must have a unique identifier, ident, which is separate from the primary key and is used publically to refer to a given source. This identifier is restricted to 32 characters at present.

Field	Type	Description	Index
id	mediumint unsigned	primary key	primary
added	datetime	date added	
description	text	source description	
notes	text	additional notes	
ident	varchar(32)	unique identifier	index

### 2.3.7 *reference*

The *reference* table holds information about the data references; typically a journal article, but any reference is supported. The format of each name in the author list consists of the surname, a comma, and then the initials without periods; the author name are separated by semi-colons. Example: Alpher, R A; Bethe H; Gamow G. The full reference (as would appear in a paper) is given in the reference field. The publication year also has its own field. A web link to the paper on the journal web site as well as the link to the NASA ADS entry can be given in the link and adslink fields, respectively. Finally, each reference is given a unique identifier. The convention for journal papers is the journal abbreviation, volume number, and page number, where the volume and page are separated by an underscore character.

Field	Type	Description	Index
id	mediumint unsigned	primary key	primary
authors	varchar(255)	author list	
year	year(4)	publication year	
reference	text	full reference	
link	varchar(255)	publication link	
ident	varchar(32)	unique identifier	index
adslink	varchar(255)	NASA ADS link	

### 2.3.8 *sourceref*

This table simply connects individual references in the *reference* table with entries in the *sources* table. One reference per source can be designated as the top reference which appears first on the webpage.

Field	Type	Description	Index
id	int unsigned	primary key	primary
sid	mediumint unsigned	source identifier	
rid	mediumint unsigned	reference identifier	
top	bit	first reference	

### 2.3.9 fundamental

The *fundamental* table defines all possible data types in the broadest sense. Examples are total photoexcitation cross section, dielectronic recombination rate, and Einstein A coefficient. While the actual data for each of these processes can come in a variety of forms, the physical process will be given by values in this table. The fundamental data type is analogous to the rate type within xstar. The value xstar uses for each rate type is stored in this table, if available. Each fundamental data type is given a unique identifier, ident, in order to publically refer to it.

Field	Type	Description	Index
id	smallint unsigned	primary key	primary
xid	smallint unsigned	xstar rate type	
label	varchar(64)	short name	
description	text	details	
ident	varchar(32)	unique identifier	
hidden	bit	hidden from user	index

### 2.3.10 datagroups

The *datagroups* table groups data types whose data values can be easily converted between group members (i.e. using formulae in the *unitconvert* table). Each data group must point to a valid fundamental data type. If a default unit is given, then this unit will be assumed if none other is given by the user in database queries. Each data group must be given a unique identifier, ident, in order to publically refer to it.

Field	Type	Description	Index
id	smallint unsigned	primary key	primary
label	varchar(64)	short name	
ident	varchar(32)	unique identifier	index
fdtid	smallint unsigned	fundamental data type	index
uid0	int unsigned	default unit	
description	text	details	

### 2.3.11 datatypes

The *datatypes* table describes all data formats used in the database. Each entry must correspond to a fundamental datatype, fdtid, and optionally an xstar data type, xid. The unit of the data values is given by uid which points to the *units*

table. The full description of the data format including all parameters (and their units) should be included in the description. The label field provides a short description of the data type for use on the web page. Each data type must be given a unique identifier, ident, in order to publically refer to it.

Field	Type	Description	Index
id	smallint unsigned	primary key	primary
fdtid	smallint unsigned	key from fundamental	index
xid	smallint unsigned	xstar data type	
label	varchar(64)	short name	
uid	int unsigned	index of units table	index
description	text	details	
ident	varchar(32)	unique identifier	index

### 2.3.12 *paramtypes*

The *paramtypes* table defines all possible parameters that data can have. The label should give the fundamental parameter type (e.g. energy) while the unit can further distinguish the parameter (e.g. Ry or eV). Some parameters, such as initial level index, have no units. As with *datatypes*, each parameter has a unique identifier which is used to publically refer to the parameter type.

Field	Type	Description	Index
id	smallint unsigned	primary key	primary
label	varchar(64)	short name	
uid	int unsigned	index of units table	index
ident	varchar(32)	unique identifier	
description	text	description	index

### 2.3.13 *paramdef*

The *paramdef* table defines the complete set of parameter types needed for each data type. This table is used to check the validity of input data sets. The table only has two columns besides the primary key; for a given data type, a row exists for each parameter needed for that type.

Field	Type	Description	Index
id	int unsigned	primary key	primary
dtid	int unsigned	key from datatypes	index
ptid	smallint unsigned	key from paramtypes	index

### 2.3.14 *setdef*

Similar to the *paramdef* table, the *setdef* table defines the parameter types for data using the *store2* or *store3* tables. The var field, either 'X' or 'Y', is used to distinguish the parameter sets.

Field	Type	Description	Index
id	int unsigned	primary key	primary
var	char	grid variable	
ptid	smallint unsigned	key from paramtypes	index
dtid	smallint unsigned	key from datatypes	index

### 2.3.15 *paramgroups*

The *paramgroups* table defines parameter groups used for sorting or converting parameters. As with *paramtypes*, each parameter group has a unique identifier which is used to publically refer to it.

Field	Type	Description	Index
id	smallint unsigned	primary key	primary
label	varchar(64)	short name	
ident	varchar(32)	unique identifier	index
uid0	int unsigned	index of units table	
description	text	description	index

### 2.3.16 *groupdef*

The *groupdef* table sets the parameter type membership for each parameter group. A parameter type can belong to more than one group.

Field	Type	Description	Index
id	int unsigned	primary key	primary
pgid	int unsigned	key from paramgroups	index
ptid	smallint unsigned	key from paramtypes	index

### 2.3.17 *data*

The *data* table stores all data not stored in the *store2* or *store3* tables. The source and type of data are given by the pointers, sid and dtid. Only the value of the data is stored in this table; all parameters associated with this value, e.g. energy or level index, is stored in the *parameters* table.

Field	Type	Description	Index
id	bigint unsigned	primary key	primary
sid	mediumint unsigned	key from sources	index
dtid	smallint unsigned	key from datatypes	index
value	varchar(32)	data value	
compare	float	value as float	

### 2.3.18 *door2*

The *door2* table contains summary information for each single-parameter dataset in the *store2* table. It also serves to connect the *store2* table to the *data* table.

Field	Type	Description	Index
id	int(10) unsigned	primary key	
did	bigint(20) unsigned	key from data	
xmin	float	smallest x-value	
xmax	float	largest x-value	
fmin	float	smallest data value	
fmax	float	largest data value	
count	int(10) unsigned	number of data points	

### 2.3.19 *store2*

Large, single-parameter datasets are stored in the *store2* table.

Field	Type	Description	Index
id	bigint unsigned	primary key	
d2id	int unsigned	key from door2	
x	varchar(32)	x value	
xcomp	float	x float value	
f	varchar(32)	data value	
fcomp	float	data float value	

### 2.3.20 *door3*

The *door3* table contains summary information for each two-parameter dataset in the *store3* table. It also serves to connect the *store2* table to the *data* table.

Field	Type	Description	Index
id	int unsigned	primary key	
did	bigint unsigned	key from data	
xmin	float	smallest x-value	
xmax	float	largest x-value	
ymin	float	smallest y-value	
ymax	float	largest y-value	
fmin	float	smallest data value	
fmax	float	largest data value	
count	int unsigned	number of data points	

### 2.3.21 *store3*

Large, two-parameter datasets are stored in the *store3* table.

Field	Type	Description	Index
id	bigint unsigned	primary key	
d3id	int unsigned	key from door3	
x	varchar(32)	x value	
xcomp	float	x float value	
y	varchar(32)	y value	
ycomp	float	y float value	
f	varchar(32)	data value	
fcomp	float	data float value	

### 2.3.22 parameters

The *parameters* table stores all parameters for the data in the *data* table. As there are typically more than one parameter per piece of data, this table will be larger than the *data* table. Pointers exist to the data value and parameter type (which contains the unit).

Field	Type	Description	Index
id	bigint unsigned	primary key	
did	bigint unsigned	key from data	
ptid	smallint unsigned	key from paramtypes	
value	varchar(32)	parameter value	
compare	float	value as float	

### 2.3.23 units

The *units* table defines all possible units. A label is provided for printing the unit to the web page, terminal, or Latex document. Each unit also has a unique identifier.

Field	Type	Description	Index
id	smallint unsigned	primary key	
ident	varchar(32)	unique identifier	
term	varchar(32)	unit for using with terminal	
web	varchar(32)	unit for using with webpage	
latex	varchar(32)	unit for using with latex	

### 2.3.24 unitalias

The *unitalias* table gives the list of alternate labels that can be used for each unit.

Field	Type	Description	Index
id	int unsigned	primary key	
uid	int unsigned	key from units	
alias	varchar(32)	alias	

### 2.3.25 *unitconvert*

The *unitconvert* table contains formulae to convert pairs of units.

Field	Type	Description	Index
id	int unsigned	primary key	primary
uto	int unsigned	key from units	index
ufrom	int unsigned	key from units	index
formula	varchar(32)	formula	

### 2.3.26 *notes*

The *notes* table allows for a note to be attached to any information in the database. There are currently two types of notes: remark and label. A remark is a comment about an individual piece of data or an entire data set; for example, if a data value is changed due to an error in the original source, a remark would be used to note this. The label type is used to provide an alternate label for a piece of data (single table row) than the internally generated label. Only one label is allowed for a table row, but there can be any number of remarks.

The *src\_table* field has the following enumerated types: 'allconfs', 'alllevels', 'allterms', 'changelog', 'data', 'datatypes', 'elements', 'fundamental', 'ions', 'log', 'notes', 'paramdef', 'parameters', 'paramtypes', 'reference', and 'sources'.

Field	Type	Description	Index
id	int unsigned	primary key	primary
src_table	enum	name of table	index
rid	bigint unsigned	key from table	index
type	enum('label','remark')	type of note	index
note	text	note	

### 2.3.27 *log*

The *log* table contains entries for when the database is modified. Only the date, change description, and name of person making the modification are stored here.

Field	Type	Description	Index
id	int unsigned	primary key	primary
date	datetime	date of change	index
name	varchar(64)	name of modifier	
msg	text	description of change	

# Chapter 3

# PHP Libraries

The PHP scripting language has been developed for the purpose of making dynamic web pages. It is long established, widely-used, and well documented. Of particular importance to this project is PHP's large set of inherent functions for interacting with mySQL databases. In this chapter we discuss the set of libraries underlying the command line scripts used to modify the database contents and the web pages used to display datasets. It is only these libraries that directly access the uaDB database.

To help organization, we have constructed object-oriented classes for each of the 27 tables in uaDB. The naming convention is *adminTbl* where *Tbl* is replaced by the database table name. Each class resides in its own file, `adminTbl.class.php`, which are in the `classes` directory. These table classes are all children of the class, `adminTable`; that is, the individual table classes inherit a common set of functions from the `adminTable` class. Also in the `classes` directory are PHP classes for each web page named *web\_page* where *page* is replaced by the web page name. These classes simply ease organizing the PHP code used to write various sections of each web page. Finally, a set of common functions reside in `lib/admin_library.php`. These functions perform a variety of tasks: categorize data and parameter types, define or access system variables, unit or type conversion, and calculate physics quantities.

## 3.1 admin\_library

This set of library functions is broken up into sections using comment characters. Here, we will only discuss briefly the type of functions each section contains. The file comments can be read for a complete description of each function.

### 3.1.1 Constants

These functions just return global constants for the database; mainly path information.

### 3.1.2 Fundamental data type categories

Currently, the database gives equal importance to all fundamental data types. However, in practice, some fundamental types are more important or more commonly used. Therefore, we have written several functions to group or prioritize the fundamental types for use on the web page. We may later modify the physical structure of the database to contain this information which would render these functions obsolete. There are no current plans for this.

### 3.1.3 Grid

These functions are used by the Grid page for populating the drop down menus and legend.

### 3.1.4 Data type categories

The functions have the same purpose as those in the Fundamental data type section except they act on data types. No more needs to be said.

### 3.1.5 Parameter group categories

Again, same as above except for parameter groups.

### 3.1.6 Parameter type categories

Again, these functions act to group or prioritize parameter types. There are many functions in this section since there are many parameters that only have small differences with each other. For example, there are several parameters indicating the initial state of a transition; a function exists to group these parameters together. There are plans to create a fundamental parameter type table which would naturally group the parameter types. This new table would make obsolete many of these functions. In the meantime, these functions (and those from the previous two sections) must be manually maintained to be up-to-date with the database contents.

### 3.1.7 System-related functions

These functions are primarily used by the command-line scripts. They help with formatting of information displayed to the command line as well as reading in argument lists.

### 3.1.8 Conversion functions

This section contains a large number of functions to convert one quantity to another. Examples include converting angular momentum letters to integers, various forms of an ion string, and roman numerals to integers.

Class	Instance	Class	Instance
adminAllConfs	\$acac	adminLog	\$loac
adminAllLevels	\$alac	adminNotes	\$noac
adminAllTerms	\$atac	adminParamdef	\$pdac
adminData	\$dac	adminParameters	\$pac
adminDatatypes	\$dtac	adminParamtypes	\$ptac
adminDoor2	\$d2ac	adminReference	\$rac
adminDoor3	\$d3ac	adminSetdef	\$sdac
adminElements	\$eac	adminSourceref	\$srac
adminErrHandler	\$errmsg	adminSources	\$sac
adminFiles	\$flac	adminStore2	\$s2ac
adminFundamental	\$fac	adminStore3	\$s3ac
adminIons	\$iac	adminUnits	\$uac

Table 3.1: Standardized instance names for each class.

### 3.1.9 Physics related functions

These are functions which contain some physics. Currently, there are functions that act on configuration and term strings.

### 3.1.10 Translation functions

These functions help to convert values between the form used by the database and the form we humans like to see.

### 3.1.11 General database functions

Often there are actions to be performed on the database which don't naturally fit in one of the table classes. These functions go into this section. Note: the functions, `get_table_list()` and `get_ident_list()` should be re-written to access the database instead of simply being a list of table names.

### 3.1.12 Web page functions

These are just functions which could be useful for all the web pages.

## 3.2 Table classes

We will not discuss all of the table classes in this section since many of them are so similar. Instead, we will detail the parent class, `adminTable`, and then go over the important child classes.

When declaring an instance of the table classes, we abide by a naming convention for consistency and to ease searching for table functions in the code. The convention is given in Table 3.1.

### 3.2.1 *adminTable*

Classes are used for the convenience of organizing functions. Therefore, no variables are associated with any of the classes. The closest thing to a variable is the *table\_name()* function giving the table name. A template of this function is in *adminTable*, but this function must be appropriately defined in all child classes. The only other functions in this class are to check if a table row exists (*finger()*) and to retrieve either a single row (*get\_row()*) or the whole table (*get\_all()*).

All child classes contain the function, *insert\_row()*, which is to be the only way data is added to the database. This function should be written into *adminTable* where the NULL status of a field determines whether it is a required or optional element. Some table classes may need to keep a customized *insert\_row()* function, namely *store2* and *store3* which insert multiple rows at once.

### 3.2.2 *adminData*

As the *data* table is one of the most important, we will spend a little time discussing the functions in its class. The majority of the searches of the database use a function in this class. The functions, *search\_by\_source()* and *search\_by\_datatype()* are used to find data types associated with a given source and vice versa. When inserting new data into the table, it is important that the data does not already exist; the *search()* function does the duplicate check. The most important functions, however, are *reduce()*, *get\_data()*, and *dataset()*. These functions are the power behind the Browser web page.

The *reduce* function is the primary searching tool and will join the necessary tables to search on fundamental data type, data type, source, reference, and any number of parameters. Furthermore, the data or parameters can be given a value or range to restrict the search. Finally, an ion string representing a single ion or sequence can be provided to focus the search further. None of these elements are required and, if none are given, the entire database will be returned. The functions are usually pretty fast, but a couple things can really slow it down. For example, querying more than 4 parameters (give or take) can severely slow down the function. For this reason, the web page is restricted to only allowing a search of three parameters simultaneously. For a larger number of parameters, it is better to use the *get\_data()* function, which we turn to now.

The *get\_data()* function retrieves data sets defined by a single source and data type. You can still restrict the search by ion string or bracket the results by data/parameter values or ranges. But since the data type and source are defined, this function is much faster than *reduce()*. When clicking on the GET, button in the results listing on the Browser page, it is this function that is called.

To retrieve data from the *store2* or *store3* table, use the *dataset()* function. The first argument of this function is the row index of the data table. Therefore, only a single dataset can gotten at once with this function. There are also arguments to allow for filtering the data on the parameters in the store2/3 tables.

The last major retrieval function is *grid\_data()* which is used by the Grid web page to get a list of data sets broken down by coupling scheme and ion. No data is actually returned, only whether or not data exists for each ion.

Finally, the appropriately named function, *delete()*, is used to delete data sets. This function will delete data simultaneously from the *data*, *parameters* tables as well as from the *door2/store2* and *door3/store3* tables, if necessary. Therefore, the classes for the other classes do not need their own *delete()* functions.

### 3.2.3 *adminIons*

The *adminIons* class is an exception in that it has some private variables. These variable define the supported element and ion set of the database. If this set is expanded beyond Zn, these variables need to be adjusted (don't forget to increase the Roman numeral array or you'll be missing some ions!). Besides element symbols, the full element names have their own array and, to cover all bases, there is an array giving the English spelling of Aluminum and Sulfur. If there are other alternate spellings missing, they should be inserted here.

Accessing these private variables are several functions dealing with elements and ions which do not actually access the database. Rightfully, these functions should belong in *admin\_library*, but they were put in this class in the initial development and never left. One day they should be moved, but carefully.

An important function in this class is *parse\_ion()* which can take an ion string in any of three forms and return a list of  $Z, z_1$  pairs where  $Z$  is the nuclear charge and  $z_1$  is the residual ion charge plus one. The three acceptable forms are chemical notation (e.g. Fe17+), astrophysical notation (Fe XVIII), and sequence notation (F-like Fe). If the charge state is left off of the first form then the iso-nuclear sequence is returned. Leaving the element off of the third form gives the iso-electronic sequence.

Also worth mentioning is the function, *label()*, which can return the ion label in any of the above forms given the row index of the ions table.

### 3.2.4 *adminErrorHandler*

A poorly-chosen name puts this class in this section; there is no *ErrorHandler* table. This class constructs a simple error handler for the actual table classes. The last argument of many functions is *\$errormsg* which is an instance of this class. It allows for an error to be passed all the way to the top level of a script for controlled termination while keeping a trace of its path. A big help for debugging. Warning messages are also supported for cases where a note to the user is desired without terminating the script. The printing routines are catered towards use in the terminal not the web page.

### 3.2.5 *adminFiles*

Another poor class name as there is no *Files* table. This class contains functions used in parsing the input files for data insertion into the database. This class is

also in charge of converting the values in the input files to the form the database expects. More details on this process are in a later chapter ([insert citation](#)). The format of the input files is explained in Appendix B.

### 3.3 Web classes

As with the table classes, we have organized web page functions into classes. For most pages, there is a class named *web\_page* where *page* is the name of the web page. However, as always, there are exceptions: *web\_generic*, *web\_info*, and *web\_pagination*.

These classes mainly contain functions to write HTML which needs very little explanation. The class, *web\_generic*, contains HTML-writing functions needed by every page as well as defining some site-wide constants. The class, *web\_pagination*, is a slightly modified version of a class found on the web (see class comments for the source) which is in charge of splitting a long list of search results into several pages. The number of results per page is set within *web\_generic*. Finally, *web\_info*, is special only in that it has HTML functions for all of the pages listing fundamental data types, data types, parameter types, and sources. No reason to have a separate class for each.

## Chapter 4

# PHP command line scripts

Modification of the database contents can only be done using a database administration tool (e.g. phpMyAdmin) or the PHP command line scripts in the `scripts` directory. All of these scripts share the prefix `xsc_` and reside in the `scripts` directory. A section will be devoted to each type of script. Using the `-h` option with all scripts will produce a help screen with no further action. Scripts that modify the database contents require the user to give his/her name which is inserted into the database log. This is done using the `-w` option.

There are also scripts which assist in documenting the database contents. These scripts have a prefix of `docgen_` and are stored in the `manuals/scripts` directory. The Python scripts are documented in a later chapter [TBD], but the PHP scripts will be discussed in this chapter.

### 4.1 `xsc_createdb.php`

When developing or debugging, it often useful to have a fresh copy of the database. This script will create all the database tables under the database name of your choosing. If you give a database name that is already in use, you must provide an addition flag allowing the script to overwrite the current version.

### 4.2 `xsc_delete.php`

As the name suggests, the script is used to delete data sets. The script can delete only one data type at a time which is the single argument to the script. The user can also specify the source, but no other filtering of the data is currently allowed. The script will remove data from the `data`, `parameters`, `door2`, `store2`, `door3`, and `store3` tables, as necessary. The specified data type and source are not removed. When run, the script checks for data with the given data type and optional source and reports the number of data rows found. To delete this data, the script must be run again but adding the `-o` option. Be careful.

Note: it is possible for a dataset to use another dataset's values as a parameter. If the latter set is deleted then the former will point to nothing. This script does **not** check for this situation, but it should. It should first check all data type definitions for parameters pointing to this data type and, if a match is found, do a detailed search for dependencies. An example is a superlevel which exists as a data type and parameter. Unfortunately, the database does not currently know when a datatype is also a parameter type making the above operation difficult.

Note: no scripts exist to delete data types, parameter types, or sources. Such actions are currently done manually using phpMyAdmin. If a such a script is written, it **must** check that no data is using the data type, parameter type, or source.

### 4.3 xsc\_optimize.php

Periodically, especially after deleting a large amount of data, it is good to optimize the tables. This allows for unused indices to be used again instead of lying dormant for perpetuity. The script should be run after any large modification to the database.

### 4.4 xsc\_browser.php

This script is a poor man's version of the Browser web page. It will allow you to perform the same filtering, but a lot of typing may be required. The one advantage this script has over the web page is its ability to search by data type in addition to fundamental data type. The data type restriction on the web page was added to avoid user confusion (secret: you can type the data type directly in the address bar to get around the restriction). The browser script can also print lists of fundamental data types, data types, parameter types, and sources. Use the `-h` option for instructions.

### 4.5 xsc\_ld\*.php

There are several scripts used for loading information into the database. All of these scripts read a standardized input file described in Appendix B. Since each input file is unique to the table it modifies, these scripts should probably be merged into a single `xsc_load.php` script. However, the reward versus effort doesn't warrant such a consolidation at this time.

### 4.6 docgen\_tabledesc.php

This simple script reads the database to extract the make-up of each table for the tables in Section 2.3. It is not a simple cut and paste operation to update

the descriptions, however, as the text description for each table is not included in the script.



# Chapter 5

## XML files

While not strictly part of the SQL database, we are using XML files to define the set of fundamental data types, data types, and parameter types. From these XML files are generated the input files for the database as well as some of the documentation you are reading now. Using XML helps ensure consistency between the manual and the database contents.

We have also recruited XML to help parse data files from their original format into the standard format used for insertion into the database (see Appendix B). To do this we define a XML file for the original data set and another XML file that connects the source XML file with the XML file defining the database data type. In the future, these XML files will also assist in writing data from the database into a variety of data formats.

In this chapter, we will define the XML files and the set of Python scripts which uses them.

Note: some characters need special treatment in XML. The most common one is the ampersand, &, in references which needs to be written as &amp;.

### 5.1 Fundamental data type definitions

The XML files defining fundamental data types are very simple. The master node for a single type is called FUNDAMENTAL and contains the following child nodes: NAME, LABEL, and DESCRIPT which give the identifier, label, and description. The single optional node, XID, stores the xstar rate type. Below is an example for partial photoionization.

```
<FUNDAMENTAL>
  <NAME>PIPAR</NAME>
  <LABEL>PI, partial</LABEL>
  <DESCRIPT>Partial photoionization.</DESCRIPT>
  <XID>7</XID>
</FUNDAMENTAL>
```

The XML code for all the fundamental data types is grouped into the file, `fundamental.xml`, under the master node, `XSTARDB_FUNDAMENTALS`.

## 5.2 Parameter type definitions

The XML files for parameter types are similar to those defining the fundamental data types. The master node for a single type is called `PARAMTYPE` and contains the following child nodes: `NAME`, `LABEL`, and `DESCRIP` which give the identifier, label, and description. The single optional node, `UNITS`, stores the parameter unit, if relevant.

```
<PARAMTYPE>
  <NAME>ENERGY_EV</NAME>
  <LABEL>energy (eV)</LABEL>
  <UNITS>eV</UNITS>
  <DESCRIP>Energy in eV.</DESCRIP>
</PARAMTYPE>
```

The XML code for all the parameter types is grouped into the file, `paramtypes.xml`, under the master node, `XSTARDB_PARAMTYPES`.

## 5.3 Data type definitions

The XML data type definitions are more complicated, so use the example below to help make sense of the following description. As with fundamental data types and parameter types, each XML file requires the nodes, `NAME`, `LABEL`, and `DESCRIP`, giving the identifier, label, and description.

To cut down on the number of definitions, the identifier, label, and description can use variables to represent the coupling scheme. There are currently six supported coupling schemes: level (IC), term (LS), conf (CA), superlevel (SL), superterm (ST), and superconf (SC). In the above list, consider each entry to be the coupling scheme's long label (short label). These variables are defined in XML file under the `VARIABLES` node. Each variable is defined by a `VARIABLE` node having child nodes of `SHORT`, `LONG`, and `DEFAULT`. The first two give the variables for the short and long labels and the last node gives the short label for the default coupling scheme. The default coupling scheme is used by some scripts using the XML files to process data. Variables should always be a single capital letter inside curly brackets. Furthermore, we are using `{A}` to represent coupling schemes for normal levels (IC, LS, and CA) and `{B}` for the super level coupling schemes (SL, ST, SC). If two normal levels with different coupling schemes are needed, we are using `{C}` for the second coupling scheme. The above variables apply to the short coupling scheme label; use `{X}`, `{Y}`, and `{Z}` for the long labels.

All parameters not to be stored in the `store2` or `store3` tables are listed under the `PARAMETERS` node. Each parameter is identified by its identifier in

a PARAMETER node. Parameters to be stored in the *store2* or *store3* tables are given in the DATASET node. Again, these parameters are in a PARAMETER node which now has the attribute, `var`, to identify whether the parameter is the X or Y variable. If only one parameter is present and the attribute is absent, X is assumed.

```

<DATATYPE>
  <NAME>PIPAR_THR_{A}{A}</NAME>
  <LABEL>PI, partial ({A}{A})</LABEL>
  <FUNDAMENTAL>PIPAR</FUNDAMENTAL>
  <XSTAR_ID>53</XSTAR_ID>
  <UNITS>Mb</UNITS>
  <DESCRIPTION>
    Partial photoionization cross section with respect to ionization
    potential of the initial state for the transition: ID_ION_INIT,
    ID_{X}_INIT to ID_ION_FINAL, ID_{X}_FINAL. The data is tabulated
    against energy in Rydbergs.
  </DESCRIPTION>
  <PARAMETERS>
    <PARAMETER>ID_ION_INIT</PARAMETER>
    <PARAMETER>ID_{X}_INIT</PARAMETER>
    <PARAMETER>ID_ION_FINAL</PARAMETER>
    <PARAMETER>ID_{X}_FINAL</PARAMETER>
  </PARAMETERS>
  <DATASET>
    <PARAMETER var='X'>ENERGY_RY</PARAMETER>
  </DATASET>
  <VARIABLES>
    <VARIABLE>
      <SHORT>{A}</SHORT>
      <LONG>{X}</LONG>
      <DEFAULT>IC</DEFAULT>
    </VARIABLE>
  </VARIABLES>
</DATATYPE>
```

The XML code for all the data types is grouped into the file, `datatypes.xml`, under the master node, `XSTARDB_DATATYPES`.

## 5.4 xstar source file

At this time, the database is solely populated from data files from the xstar modeling code<sup>1</sup>. Each xstar file is defined by its xstar data type (not to be confused with our database data types). The XML files described here can

---

<sup>1</sup><http://heasarc.nasa.gov/lheasoft/xstar/xstar.html>

describe nearly all of the data files for xstar. The exceptions are xstar data types 6, 66, and 78.

Before getting to the XML file, a brief description of the xstar file format is useful. Each line in an xstar file represents a data set. The first six numbers are integers describing the format of the rest of the line. The first two integers give the xstar data type and rate type. This is followed by a number whose meaning has been lost to time, but is always zero. The last three numbers give the number of floating point values, integer values, and length of the comment string. Then come the floating point values, integer values, and comment string which cannot have any spaces. An xstar line is always terminated with the % symbol (a single xstar data line can therefore span several physical lines in the file). There are no assumed fixed formats for files, instead a space-delimiter is used to distinguish values. Below is an example of the XML file having a master node of **XFILE** followed by a discussion.

```

<XFILE>
  <DID>50</DID>
  <RIDS>
    <RID>4</RID>
    <RID>9</RID>
  </RIDS>
  <NREAL>3</NREAL>
  <NINTEGER>4</NINTEGER>
  <DESCRIPTION>line rad. rates from OP and IP</DESCRIPTION>
  <COMMENT>
    Transition probabilities file. Every line contains: reals (3;
    Wavelength (A), gf-val., A-val. (cm-1)) ; integers (4; lower level,
    upper level, Z, $\\#ion) ; characters (0)
  </COMMENT>
  <VALUES>
    <VALUE units="Angstrom">
      <LABEL>wavelength</LABEL>
      <POS>R1</POS>
      <HELP>wavelength (Angstroms)</HELP>
    </VALUE>
    <VALUE>
      <LABEL>gfvalue</LABEL>
      <POS>R2</POS>
      <HELP>gf-value</HELP>
    </VALUE>
    <VALUE units="1/s">
      <LABEL>avalue</LABEL>
      <POS>R3</POS>
      <HELP>Einstein $A$ coefficient (1/s)</HELP>
    </VALUE>
    <VALUE desc="levl">
```

```

<LABEL>lev_lower</LABEL>
<POS>I1</POS>
<HELP>lower level index</HELP>
</VALUE>
<VALUE desc="levu">
  <LABEL>lev_upper</LABEL>
  <POS>I2</POS>
  <HELP>upper level index</HELP>
</VALUE>
<VALUE desc="z">
  <LABEL>z</LABEL>
  <POS>I3</POS>
  <HELP>nuclear charge</HELP>
</VALUE>
<VALUE desc="ion">
  <LABEL>ion</LABEL>
  <POS>I4</POS>
  <HELP>xstar ion number</HELP>
</VALUE>
</VALUES>
</XFILE>

```

The xstar data type is identified by the DID node and the xstar rate types (there can be more than one), are identified by RID nodes which are children of the RID node. The number of reals and integers are given by NREAL and NINTEGER nodes. If these numbers do not agree with what is found in the file then an error is generated. Some files have a variable number of real numbers; use  $-1$  for the number of reals in this case. A description of the file is contained in the DESCRIPTION node and further comments can be made in COMMENT.

The rest of the file consists of defining the values and resides in the VALUES tag. Not all values need to be defined in the XML file, only those you plan to extract. Each value is defined by a VALUE node containing the child nodes: LABEL, POS, and HELP. If the value has a well-defined unit, this should be defined using the attribute, units, in the VALUE tag. The VALUE tag also can have the attribute, desc, which declare that the value is a well-defined type, such as nuclear charge or level index. The full list of these descriptors is in Table 5.1. The LABEL node is used to identify the value in the interface XML file (see next section) while the HELP node is used for documentation. The tag, POS, identifies the position of the data in the file. To retrieve the comment data, use a POS of comment. A single integer or real number can be specified with I# or R# where # represents the ordinal number. Finally, a range of reals can be collectively referenced using the range notation, R(*start,end,step*) where *start*, *end*, and *step* are expressions involving integers, normal arithmetic operators, and variable representing the total number of real columns (N) or integer columns (I#). For example if the real data represents several  $x,y$  pairs and you want to reference  $x$ , you could use R(1,N-1,2). On the other

descriptor	description
ion	ion index of current ion
ionp	ion index of plus ion
lev	level index of current ion
levp	level index of plus ion
slev	super level index of current ion
slevp	super level index of plus ion
levl	lower level index
levu	upper level index
n	principal quantum number
z	nuclear charge
l	L
g	2S+1
w	2J+1; weight
j2	2J

Table 5.1: Descriptors for XML file describing xstar source files.

hand, if the source file lists all  $x$  values followed by all the  $y$  values, the **POS** tag would take **R(1,I1)** where I1 would point the number of  $x$  values. Note, in this last example, **step** has been omitted since it is 1.

For data existing on a lattice (dimension > 2), often the parameter sets are given as consecutive 1D arrays preceding the data. In order to construct the full data set correctly, we use a **GRID** tag with the **VALUE** section. The **GRID** tag takes an attribute of *name* whichs labels the grid and has a integer value. Each parameter set defining the grid will have a **GRID** tag with the same name; the parameter with the smallest grid value changes the quickest over the data set, followed by the next smallest, etc. Below is a snippet of XML showing the real number values for a xstar file containing first an array of densities, followed by an array of temperatures, and then the two-dimensional array of recombination rates. This is all followed by pairs of points representing energy and photoionization cross sections. The integer values, I1 and I2, give the number of densities and temperatures. It is about as complicated as the xstar files get.

```
...
<VALUE units="cm^3/s">
  <LABEL>density</LABEL>
  <POS>R(1,I1,1)</POS>
  <GRID name="dentemp">2</GRID>
  <HELP>density grid ($1/cm^3$)</HELP>
</VALUE>
<VALUE units="K">
  <LABEL>temperature</LABEL>
  <POS>R(I1+1,I1+I2,1)</POS>
  <GRID name="dentemp">1</GRID>
```

```

<HELP>temperature grid (K)</HELP>
</VALUE>
<VALUE units="1/s">
  <LABEL>recombination</LABEL>
  <POS>R(I1+I2+1,I1+I2+I1*I2,1)</POS>
  <HELP>recombination rate array (1/s)</HELP>
</VALUE>
<VALUE units="Ry">
  <LABEL>energy</LABEL>
  <POS>R(I1+I2+I1*I2+1,N-1,2)</POS>
  <HELP>energy grid (Ry)</HELP>
</VALUE>
<VALUE units="Mb">
  <LABEL>pics</LABEL>
  <POS>R(I1+I2+I1*I2+2,N,2)</POS>
  <HELP>cross section array (Mb)</HELP>
</VALUE>
...

```

A final note: all subscripts and superscripts in the DESCRIPTION, COMMENT, and HELP tags should written as L<sup>A</sup>T<sub>E</sub>X would expect; that is, use \$.

## 5.5 XML interface

As you might imaging, the XML format for interfaces connecting the xstar source files with the database data types can be rather complicated. Below is a sample file containing two interfaces.

```

<XSTARDB_INTERFACES>
  <INTERFACE>
    <DATATYPE>RRTOT_ALDPEQ_{A}</DATATYPE>
    <DID>1</DID>
    <RID>8</RID>
    <DATA_CONNECT>
      <SUB>string:</SUB>
    </DATA_CONNECT>
    <PARAM_CONNECTS>
      <PARAM_CONNECT>
        <PTYPE>ID_ELEMENT</PTYPE>
        <LABEL>ion</LABEL>
        <XFORM>ion2elem</XFORM>
      </PARAM_CONNECT>
      <PARAM_CONNECT>
        <PTYPE>ID_{X}_INIT</PTYPE>
        <SUB>ground:ion</SUB>
      </PARAM_CONNECT>
    </PARAM_CONNECTS>
  </INTERFACE>

```

```

<PARAM_CONNECT>
  <PTYPE>PARAM_1</PTYPE>
  <LABEL>r1</LABEL>
</PARAM_CONNECT>
<PARAM_CONNECT>
  <PTYPE>PARAM_2</PTYPE>
  <LABEL>r2</LABEL>
</PARAM_CONNECT>
</PARAM_CONNECTS>
<CHECKS>
  <CHECK type="constant">ion</CHECK>
</CHECKS>
</INTERFACE>

<INTERFACE>
  <DATATYPE>WAVELEN_{A}{A}</DATATYPE>
  <DID>50</DID>
  <RID>4</RID>
  <DATA_CONNECT>
    <LABEL>wavelength</LABEL>
  </DATA_CONNECT>
  <PARAM_CONNECTS>
    <PARAM_CONNECT>
      <PTYPE>ID_ELEMENT</PTYPE>
      <LABEL>ion</LABEL>
      <XFORM>ion2elem</XFORM>
    </PARAM_CONNECT>
    <PARAM_CONNECT>
      <PTYPE>ID_{X}_INIT</PTYPE>
      <LABEL>lev_upper</LABEL>
    </PARAM_CONNECT>
    <PARAM_CONNECT>
      <PTYPE>ID_{X}_FINAL</PTYPE>
      <LABEL>lev_lower</LABEL>
    </PARAM_CONNECT>
  </PARAM_CONNECTS>
  <CHECKS>
    <CHECK type="constant">ion</CHECK>
  </CHECKS>
</INTERFACE>

```

Only one interface can exist for each combination of database data type, DATATYPE; xstar source data type, DID; and xstar source rate type, RID. The tags, DATA\_CONNECT and PARAM\_CONNECT, describe how the xstar source values relate to the database data and parameter types. The parameter types are given with the PTYPE tags (since there is only one data type, an extra tag is

value	description
string:	no data
ground:ion	take ground level from given ion

Table 5.2: Supported values of SUB tag in interface XML file.

not needed), and the source values are given using LABEL tags whose values correspond to the LABEL tags in the xstar source XML file. In some cases, the data values need to undergo some transformation to be ready for the database. The XFORM tag can provide a code to describe the necessary transformation. Another way to specify a transformation is using the SUB tag in place of LABEL. The value for this tag must take the form of *vtype:label* where *vtype* describes the final form the value will take and *label* is what normally goes in LABEL. Currently supported values for SUB are in Table 5.2.

Note: XFORM and SUB tags could probably be consolidated or removed entirely.

### 5.5.1 Examples for PARAM\_CONNECT tags

The simplest parameter connection gives the database data type and the xfile label:

```
<PARAM_CONNECT>
<PTYPE>PARAM_1</PTYPE>
<LABEL>r1</LABEL>
</PARAM_CONNECT>
```

Sometimes the value in the xfile needs to be transformed before going into the database; the XFORM tag allows for this:

```
<PARAM_CONNECT>
<PTYPE>ID_ELEMENT</PTYPE>
<LABEL>ion</LABEL>
<XFORM>ion2elem</XFORM>
</PARAM_CONNECT>
```

Here, instead of taking an ion string, we use XFORM to grab just the element symbol.

For some datatypes, a parameter is needed which is not explicitly stored in the xfile. There are two ways to handle these cases. If the value needed is implicitly stored in the database, we can use the SUB tag to substitute one kind of data for another (the transformation will then be done by the data loading script). Below is an example where we want the ground state of an ion and therefore take the ion label from the xfile:

```
<PARAM_CONNECT>
<PTYPE>ID_{X}_INIT</PTYPE>
```

```
<SUB>ground:ionp</SUB>
</PARAM_CONNECT>
```

The form of the substitution is *vtype:label* where *vtype* is the value type to use in the database input file and *label* is what normally goes into the LABEL tag. Note the the SUB tag is also used in DATA\_CONNECT for parameterized data sets:

```
<DATA_CONNECT>
  <SUB>string:</SUB>
</DATA_CONNECT>
```

To explicitly give the value for a parameter, we use the VALUE tag. The value of the VALUE tag is *type:value* where *type* informs the script of what kind of value is being given and *value* is the value you want to have. For example, in charge exchange data sets, since the projectile is not stored in the xfile, we need to set it in the interface:

```
<PARAM_CONNECT>
  <PTYPE>ID_ION_PROJ</PTYPE>
  <VALUE>ion:H0+</VALUE>
</PARAM_CONNECT>
```

Finally, there are times where a single parameter value needs to be derived from multiple values in the xfile. The most common example is for sub-shells. In xfiles, the sub-shell is given by two integers, *n* and *l*, whereas for the database, a single *nl* string is stored. Using multiple LABEL tags with the appropriate XFORM tag is all that is needed for these situations:

```
<PARAM_CONNECT>
  <PTYPE>SUBSHELL</PTYPE>
  <LABEL>n</LABEL>
  <LABEL>l</LABEL>
  <XFORM>nl2subshell</XFORM>
</PARAM_CONNECT>
```

## 5.6 Python code reference

In this section, we will briefly outline the Python classes that read and organize the data in the XML files. In Figure 5.1 is a schematic of the Python classes used to handle all of the interfaces. The classes on the left handle the database information and those on the right deal with the xstar source files. The classes in the center define the interface.

First, the database classes. For each datatype, an instance of `xdb_datatype` is created; all of these instances are stored in `xdb_datatype_list`. The class, `xdb_variable`, stores each coupling variable and handles the transformation to a particular coupling scheme.

The classes dealing with the xstar source files have a similar hierarchy. The list of all xstar datatypes is contained in `xdb_xfile_list` where the details of each file are stored in an instance of `xdb_xfile`. Since the data values can be quite complicated, the `xdb_value` class has been constructed to ease their manipulation.

Finally, an instance of `xdb_datatype` can be connected to an instance of `xdb_xfile` using the `xdb_interface` class. All of the interfaces are stored in `xdb_interface_list`.

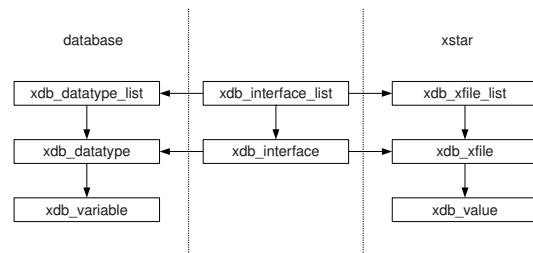


Figure 5.1: Schematic of Python classes dealing with database/xstar interface.



# Chapter 6

## Creating and populating UADB

This chapter describes how to use command-line scripts to create the UADB database from scratch and populate it with xstar data. It is recommended that all of the commands listed in this section are performed from the *ins* directory.

### 6.1 Creating the MySQL database

The script, `xsc_makeuadb.php`, will create the UADB database and all of the tables. Assuming you are in the *ins* directory, do:

```
$ ./scripts/xsc_makeuadb.php
```

Afterwards you should be able to see a database called, *uadb*, with 21 tables.

### 6.2 Add definitions and types

The next step is to add the special parameter definitions (e.g. ions, levels) and data/value types. The script, `xsc_fillbase.py`, accomplishes this task when using the `--all` option (otherwise you can fill tables individually with the appropriate option, use `--help`). Do:

```
$ ./scripts/xsc_fillbase.py -n "Your Name" --all
```

to insert units, unit conversion formulae, elements, ions, state info (configurations, terms, and levels), data types, data sub-types, value types, value sub-types, and references. The UADB input files for all of these must be already be prepared and sit in the appropriate sub-directory of *inputs*. The input files for data/value (sub-)types are generated from XML files, see Section 5. The input files for units (and their conversions), elements, and ions are premade and probably do not need to change much (although it is trivial to modify them).

Similarly, a large set of electronic states have been pre-generated. However, if more states need to be added, you can use the `levprep.py` script described in Appendix ??.

## 6.3 Adding xstar data

With a few exceptions, all xstar data is inserted in the same way. Since all xstar data refer to level indices defined in the type 6 files, we will first describe how to insert that data. Then, the general prescription for adding any data type is given, followed by the list of exceptions and notes for specific data types.

A note about super levels. All super level data must have the same source in order for the parsing scripts to understand what is going on. The type 6 script has the super level source (XSUPER) hardcoded and this source must be used for all other data involving super levels.

### 6.3.1 Type 6

The xstar type 6 files are the first data to add. The script, `xsc_parse006.py`, will read the xstar file and write UADB input files (`pout.*`) as well as a mapping file. The mapping file links the level index with its coupling scheme and unique label. The file will have the same name as the xstar file, but with an extension of `.map` and needs to be available for other parsing scripts to read (i.e. keep these in the `ins` directory). A UADB input file will be created for each coupling scheme found in the file and another containing super level data. While you can choose the source for the regular data, the super level data has a hard-coded source of XSUPER. If a file contains multiple coupling schemes, a warning will be written to screen and a file ending with `.xx` will be created. This file will contain states where the coupling scheme is ambiguous. If you are using newer xstar files with unique labels (see Appendix ??), then this file should be empty, but it is always good to check. If it is not empty, then you need to decide what the coupling scheme is for each state and copy that line into the appropriate `pout.*` file. Be sure to delete the `*.xx` files before populating the database.

Once ready, the `pout.*` files can be loaded into the database using the script, `xsc_ldfile.py`. Since there are many type 6 files, scripts have been created to run `xsc_parse006.py` on each file found and then load the UADB input files. Before doing this, though, it is a good idea to first load the sources into the database. Assuming you are in the `ins` directory and the type 6 files have been copied across, here are the steps to take:

```
$ ./scripts/xsc_ldfile -w "Your Name" xstar_levels.source
$ ./scripts/xsc_ldfile -w "Your Name" xsuper.source
$ ./scripts/xsc_qpase.py 6
$ ./scripts/xsc_qload.py "Your Name"
$ mkdir loaded
$ ./scripts/xsc_movepout.py <dirname>
```

The `-w` option in the first two lines just tells the database who is inserting the data; change this to your name. The last two lines allow for the `pout.*` files to be stored in case you want to repopulate UADB without parsing all the xstar files again. The script, `xsc_movepout.py`, will store the files in directories named by the source. It is recommended that you also store the `*.source` files here.

Notes:

- if you have a empty line in the xstar file, `xsc_parse006.py` will give an error; just remove the blank line and try again
- Sc-like Fe (d331) has a placeholder level labeled ‘fake’ for historical reasons; this needs to be removed before running the above scripts

### 6.3.2 General insertion procedure

Here, we describe the general procedure for adding xstar data to UADB. Exceptions will be described in the following subsections by xstar type. The procedure closely follows what is done with type 6 files, except that a different parsing script is used: `xsc_parsex.py`. This script is called from within `xsc_qparse.py` which will generate UADB input files from all xstar files found of the given type. At the top of the script, the sources are defined for each ion and data type. You should look in this file to see which sources need to be added to UADB prior to inserting the data, and perhaps make modifications if new data are being added.

We begin by copying the necessary source inputs (`*.source`) and all of the xstar files of a given type into the `ins` directory (which should also contain the mapping files generated in the previous subsection). You will also need to copy or link the XML files from the `xml` directory. The parsing script uses these files to know what to expect for each data sub-type. Then run the following commands:

```
$ ./scripts/xsc_ldfile -w "Your Name" <source>
$ ./scripts/xsc_qparse.py <xid>
$ ./scripts/xsc_qload.py "Your Name"
$ ./scripts/xsc_movepout.py <dirname>
```

Repeat the first command for every source you have and replace `xid` with the integer representing the xstar data type. Also, after completing the above, be sure to copy the source files into the appropriate subdirectories in `<dirname>`. Finally, you can remove the xstar files from the `ins` directory to make room for the next set.

### 6.3.3 type 50

Some xstar type 50 files contain transitions from super levels. This cannot be handled by the parsing script in its present form, but another script has

been created to extract the super level transitions from the `pout.*` files written by `xsc_qparse.py` and put them into new `pout.*` files. So, after running `xsc_qparse.py`, run `t50parse.py` and then proceed to `xsc_qload.py` as normal.

#### 6.3.4 type 66

The parsing scripts cannot handle the data in the type 66 files, so these files need to be created manually. The UADB data formats are the same as produced for type 69 files: `EIEPAR_KATNAK_{A}{B}` and `ETRANS_EV_{A}{B}`.

#### 6.3.5 type 71

This data type contains A-values for super levels, but the file for  $\text{Fe}^{23+}$  (d349) also contains A-values to the super level. These data need to be manually moved to a new `pout.*` file where the initial and final coupling schemes are reversed.

# Chapter 7

## How-to Guide

This chapter contains detailed tutorials explaining the guts of various scripts or how to expand the database functionality.

### 7.1 Adding special parameter types

New parameter types are added to the database following the instructions in Section ???. For most parameters this is enough, however there are some parameters where the value stored in the database is different than what a user would normally use. We will refer to these as special parameters and discuss the extra effort needed to make these parameters user-friendly.

When converting all parameters from user input to the database value, the function, `adminFiles→parse_value()`, is used. This function uses something called the value type to decide how to perform the conversion, but the parameter type can also indirectly affect the process. The value types for non-special parameters are self-explanatory: (int), (float), and (string). Other value types, such as (ion) or (level), tell the function to expect the parameter value to have a specific format. With a value type of (ion), the user can identify an ion using any of the following three forms: chemical (e.g. Fe20+), astrophysical (e.g. Fe XXI), and sequence (C-like Fe). In this example, the conversion is not straightforward, so `parse_value()` calls the function `process_ion()` in `admin_library.php` to do the actual conversion. Other `process_*`() functions are used when the conversion is not trivial. While the value type takes care of most of the special parameters, there are cases when the parameter type is also needed. We try to keep such occurrences to a minimum and never refer to any parameter types directly in `parse_value()`. Instead, we allow the function to inquire about properties of the parameter type using functions in the *Parameter and data type categories* section of `admin_library.php`. This approach is needed, for example, when giving the nuclear charge of an element. The value type is integer, but we need `parse_value()` to return the proper index from the `elements` table. In this case the function, `ptype_element()`, in `admin_library.php` is called which returns

true if the parameter type refers to an element. So, if you wanted to add a new parameter type referring to an element, *pctype\_element()* would need to be modified to return true when passed the new parameter type. More discussion about value types in relation to inserting new data can be found in Section ???. A table of all supported value types can also be found there (Table B.1).

We also need methods which will take the parameter value stored in the database and convert it into a form which is understood by the user. The primary function to retrieve any data from the database is *adminData→reduce()*, therefore it has been tasked with converting all special parameter values to their user-friendly form. At this time, some parameter types are explicitly used in *reduce()*. While not an ideal solution, it was expedient and only used for a small number of parameter types. Other parameter types are referred to indirectly using the same functions in *admin\_library* discussed above. All of the logic for seeking and processing the special parameter types is contained in *reduce()*, therefore this function may need to be modified when adding new special parameter types. The actual conversion of the value should always occur using a function located in *admin\_library.php*. Ultimately, the conversion logic should take place outside the main *reduce()* function.

Finally, the function, *adminFiles→parse\_parameter\_format()* needs to have the new value type added. This function is responsible for interpreting the parameter lines in the standardized input files for inputting data into the database.

## 7.2 Processing PREP\_DATA input files

The script, `xsc_lddata.php`, is used to load data into the database. The input files for this script are standardized according to the rules in Section ?? with a file type of PREP\_DATA. The processing of the input file involves three steps.

The first step is a call to *adminFiles→read\_file()* which reads the entire input file and returns an associative array with three keys: `file_type`, `file_header`, and `file_data`. The first key holds a string with the file type (should be PREP\_DATA in this case). The second key holds another associative array with the header contents. The structure of each header line is defined as `KEY: VALUE` which is matched by the structure of the `file_header` array; the keys are named KEY and have values of VALUE. If there exists more than one line with a given KEY, the array value is itself an array (not associative) containing all the VALUES. Finally, the `file_data` key is a non-associative array containing the data lines following the header information in the file.

Once the input file is read, the header information needs to be processed in preparation for reading the data. The function, *adminFiles→process\_header*, takes the output array from the previous step and returns four arrays: `$pinfo`, `$linfo`, `$vpinfo`, and `$vlinfo`. The first is an associative array mapping the parameter types (keys) to their value types (values). The list of parameter types must match exactly the definition of the data type of the data in the file. Therefore, the parameter type must exist in the database (see Table ??) and the value types, which describe how the parameter data is given in the file, must be one of

those given in Table B.1. The second output, `$linfo`, has the same form as `$vinfo` but allow for alternate labels or general remarks to be provided for each parameter. For more information about labels and remarks see Section ???. The final two output arrays, `$vinfo` and `$vlinfo`, contain options supplied for each value type. Again, see Section ??? for more information. Each parameter line (and the DATATYPE line) is parsed using `adminFiles→process_parameter_format()` which has each value type hard-coded. If a column of data needs to be mapped to data in another file, it is this function that retrieves the mapping array from the other file using `adminFiles→get_mapping()`.

Using the four arrays described above, the data lines can finally be processed. Each line is passed into `adminFiles→process_line()` which will separate the line into a list of values (using spaces as the delimiter) and call `adminFiles→process_value()` for each value. The parameter and value types are passed into this last function to properly convert the value into the form suitable for the database. The value types are hard-coded into `process_value()` but the parameter types are not. Instead, functions of the form `ptype_*`() are used to see if a given parameter type satisfies certain requirements (e.g. points to the allterms table). The parameter types are hard-coded in the `pype_*`() functions. The function, `process_line()`, will return an associative array whose keys are the parameter types (or data type) for each column and the values are the processed data values for each parameter. If a parameter points to a table in the database (e.g. ion or level), then the processed values are the primary keys for the appropriate table. Therefore, these values are ready for insertion into the database.

After the data is retrieved, a check is made to see if it already exists in the database (by parameter values, not data value). If *all* data is new to the database, then it is inserted and a log entry is made.



# Appendix A

## xstar data types

In this appendix, each xstar data type supported by the database are outlined and what xstardb datatypes are used to store the data. In the identifiers for the xstardb data types, sometimes a {X} or {A} is used. These indicate different coupling schemes that the levels can have. The {X} and {Y} variables can be CONF, TERM, or LEVEL while {A} and {B} can take CA, LS, IC.

For each data type, the description and comment from the xstar manual is given followed by a supplementary description and a listing of the values in the data file. Then, there is a table showing the mapping from the xstar data file to the database.

### A.1 Type 1

Datatype: 1

Ratetypes: 8

Description: Parameterized formula for total radiative recombination rate coefficients given by Aldrovandi & Pequignot (A&A 25, 137, 1973).

Comment: rate=r1/T\*\*r2

Value Description: R1: parameter, r1; R2: parameter, r2; I1: xstar ion number

Mapping:

	Data/Parameter type	mapping
datatype	RRTOT_ALDPEQ_{A}	
parameter	ID_ION_INIT	I1
parameter	PARAM_1	R1
parameter	ID_{X}_INIT	I1
parameter	PARAM_2	R2

## A.2 Type 2

Datatype: 2

Ratetypes: 8, 5

Description: Charge exchange with  $H^0$ , from Kingdon & Ferland:

Comment: rate=aax\*expo(log(t)\*bbx)\*(1.+ccx\*expo(ddx\*t))\*(1.e-9); ans1=rate\*xh0  
; where xh0 is the number density of neutral hydrogen.

Value Description: R1: parameter, aax; R2: parameter, bbx; R3: parameter,  
ccx; R4: parameter, ddx; R5:  $T_{min}$  (K); R6:  $T_{max}$  (K); R7: not used;  
I1: xstar ion number

Mapping:

	Data/Parameter type	mapping
datatype	CXPAR_KINFER_{A}{A}	
parameter	TEMP_K_MAX	R6
parameter	ID_ION_PROJ	'H0+'
parameter	ID_ION_INIT	I1
parameter	PARAM_1	R1
parameter	PARAM_3	R3
parameter	PARAM_2	R2
parameter	ID_ION_FINAL	I1
parameter	ID_{X}_INIT	I1
parameter	ID_{X}_FINAL	I1
parameter	PARAM_4	R4
parameter	TEMP_K_MIN	R5
datatype	CXTOT_KINFER_{A}	
parameter	TEMP_K_MAX	R6
parameter	ID_ION_PROJ	'H0+'
parameter	ID_ION_INIT	I1
parameter	PARAM_1	R1
parameter	PARAM_3	R3
parameter	PARAM_2	R2
parameter	ID_{X}_INIT	I1
parameter	PARAM_4	R4
parameter	TEMP_K_MIN	R5

Note: last real value is not used

## A.3 Type 7

Datatype: 7

Ratetypes: 8

Description: Parameterized formula for total dielectronic recombination rate coefficients given by Aldrovandi & Pequignot (A&A 25, 137, 1973).

Comment:  $\text{rate} = r1 * 10^{**(-6)} * e^{**(-r2/T)} * (1 + r3 * e^{**(-r4/T)}) / T^{**3/2}$

Value Description: R1: parameter, r1; R2: parameter, r2; R3: parameter, r3;  
R4: parameter, r4; R5: not used; I1: xstar ion number

Mapping:

	Data/Parameter type	mapping
datatype	DRTOT_ALDPEQ_{A}	
parameter	PARAM_4	R4
parameter	ID_ION_INIT	I1
parameter	PARAM_1	R1
parameter	PARAM_3	R3
parameter	PARAM_2	R2
parameter	ID_{X}_INIT	I1

Note: Number of reals is inconsistent; some files have 4 others have 5. The 5th value is never used.

## A.4 Type 10

Datatype: 10

Ratetypes: 5, 15

Description: Charge exchange with  $H^+$ , formula from Kingdon & Ferland.

Comment:  $\text{rate} = aax * t^{**bbx} * (1 + ccx * \exp(ddx * t)) * \exp(-eex/t) * (1.e-9); ans1 = rate * xh1$

Value Description: R1: parameter,  $a$ ; R2: parameter,  $b$ ; R3: parameter,  $c$ ; R4:  
parameter,  $d$ ; R5:  $T_{min}$ ; R6:  $T_{max}$ ; R7: parameter,  $\Delta E$ ; R8: not used;  
I1: xstar ion number

Mapping:

	Data/Parameter type	mapping
datatype	CXTOT_KINFER_{A}	
parameter	TEMP_K_MAX	R6
parameter	ID_ION_PROJ	'H1+',
parameter	ID_ION_INIT	I1
parameter	PARAM_1	R1
parameter	PARAM_3	R3
parameter	PARAM_2	R2
parameter	ID_{X}_INIT	I1
parameter	PARAM_4	R4
parameter	TEMP_K_MIN	R5
datatype	CXPAR_KINFER_{A}{A}	
parameter	TEMP_K_MAX	R6
parameter	ID_ION_PROJ	'H1+',
parameter	ID_ION_INIT	I1
parameter	PARAM_1	R1
parameter	PARAM_3	R3
parameter	PARAM_2	R2
parameter	ID_ION_FINAL	I1
parameter	ID_{X}_INIT	I1
parameter	ID_{X}_FINAL	I1
parameter	PARAM_4	R4
parameter	TEMP_K_MIN	R5

## A.5 Type 16

Datatype: 16

Ratetypes: 5, 15

Description: not used

Comment:

Value Description: R('1', 'N-4', '5'): ionization potential; R('2', 'N-3', '5'): parameter, *A*; R('3', 'N-2', '5'): parameter, *B*; R('4', 'N-1', '5'): parameter, *C*; R('5', 'N', '5'): parameter, *D*; I('2', 'N-2', '2'): sub-shell *n*; I('3', 'N-1', '2'): sub-shell *l*; I('N', 'N', '1'): xstar ion number; I1: number of sub-shells

Mapping:

	Data/Parameter type	mapping
datatype	EIITOT_ARNRAY_{A}	
parameter	SUBSHELL	I('2', 'N-2', '2'), I('3', 'N-1', '2')
parameter	PARAM_5	R('5', 'N', '5')
parameter	PARAM_4	R('4', 'N-1', '5')
parameter	ID_ION_INIT	I('N', 'N', '1')
parameter	PARAM_1	R('1', 'N-4', '5')
parameter	PARAM_3	R('3', 'N-2', '5')
parameter	PARAM_2	R('2', 'N-3', '5')
parameter	ID_{X}_INIT	I('N', 'N', '1')
datatype	EIIPAR_ARNRAY_{A}{A}	
parameter	SUBSHELL	I('2', 'N-2', '2'), I('3', 'N-1', '2')
parameter	ETHRESH_EV	R('1', 'N-4', '5')
parameter	ID_ION_INIT	I('N', 'N', '1')
parameter	PARAM_1	R('2', 'N-3', '5')
parameter	PARAM_3	R('4', 'N-1', '5')
parameter	PARAM_2	R('3', 'N-2', '5')
parameter	ID_ION_FINAL	I('N', 'N', '1')
parameter	ID_{X}_INIT	I('N', 'N', '1')
parameter	ID_{X}_FINAL	I('N', 'N', '1')
parameter	PARAM_4	R('5', 'N', '5')

## A.6 Type 25

Datatype: 25

Ratetypes: 5, 15

Description: Collisional Ionization data from Raymond &amp; Smith:(xstar1)

Comment: r1=e ; r2=a ; r3=b ; r4=c ; r5=d; ch = 1./chi; fchi = 0.3\*ch\*(a+b\*(1.+ch)+(c-(a+b\*(2.+ch))\*ch)\*alpha+d\*beta\*ch); rate = 2.2e-6\*sqrt(chir)\*fchi\*expo(-1./chir)/(e\*sqrt(e))

Value Description: R1: parameter, e; R2: parameter, a; R3: parameter, b; R4: parameter, c; R5: parameter, d; I1: initial level index; I2: xstar ion number

Mapping:

	Data/Parameter type	mapping
datatype	EIITOT_RAYSMI_{A}	
parameter	PARAM_5	R5
parameter	PARAM_4	R4
parameter	ID_ION_INIT	I2
parameter	PARAM_1	R1
parameter	PARAM_3	R3
parameter	PARAM_2	R2
parameter	ID_{X}_INIT	I1
datatype	EIIPAR_RAYSMI_{A}{A}	
parameter	PARAM_5	R5
parameter	ID_{X}_FINAL	I2
parameter	ID_ION_INIT	I2
parameter	PARAM_1	R1
parameter	PARAM_3	R3
parameter	PARAM_2	R2
parameter	ID_ION_FINAL	I2
parameter	ID_{X}_INIT	I1
parameter	PARAM_4	R4

## A.7 Type 30

Datatype: 30

Ratetypes: 8

Description: Parameterized formula for total radiative recombination rate coefficients by Gould & Thakur (Ann. Phys. 61, 351, 1970).

Comment:  $rate = 2*(2.105 \times 10^{-22})*vth*y*\phi$  where: zeff=r1; beta=zeff\*zeff/(6.34\*t6); yy=beta; vth=(3.10782e+7)\*sqrt(t); fudge factor makes the 2 expressions join smoothly; ypow=min(1.,(0.06376)/yy/yy) fudge=1.\*(1.-ypow)+(1./1.5)\*ypow phi1=(1.735+alog(yy)+1./yy)\*fudge/2. phi2=yy\*(-1.202\*alog(yy)-0.298) phi=phi1 if (yy.lt.0.2525) phi=phi2

Value Description: I1: nuclear charge; I2: xstar ion number

Mapping:

	Data/Parameter type	mapping
datatype	RRTOT_GOUTHA_{A}	
parameter	ID_ION_INIT	I2
parameter	ID_{X}_INIT	I2

## A.8 Type 38

Datatype: 38

Ratetypes: 8

Description: total dr from badnell (mcw: really rr)

Comment:

Value Description: R1: parameter,  $A$  ( $\text{cm}^3/\text{s}$ ); R2: parameter,  $B$ ; R3: parameter,  $T_0$  (K); R4: parameter,  $T_1$  (K); R5: parameter,  $C$ ; R6: parameter,  $T_2$  (K); I1: nuclear charge; I2: number of electrons; I3: initial level index; I4: initial level weight; I5: xstar ion number

Mapping:

	Data/Parameter type	mapping
datatype	RRTOT_BADNELL_{A}	
parameter	PARAM_5	R5
parameter	PARAM_4	R4
parameter	ID_ION_INIT	I5
parameter	PARAM_6	R6
parameter	PARAM_1	R1
parameter	PARAM_3	R3
parameter	PARAM_2	R2
parameter	ID_{X}_INIT	I5

## A.9 Type 39

Datatype: 39

Ratetypes: 8

Description: total rr from badnell (mcw: really dr)

Comment:

Value Description: R1: parameter,  $c_1$  ( $\text{cm}^3/\text{s K}^{3/2}$ ); R10: parameter,  $T_5$ (K); R11: parameter,  $c_6$  ( $\text{cm}^3/\text{s K}^{3/2}$ ); R12: parameter,  $T_6$ (K); R13: parameter,  $c_7$  ( $\text{cm}^3/\text{s K}^{3/2}$ ); R14: parameter,  $T_7$ (K); R2: parameter,  $T_1$ (K); R3: parameter,  $c_2$  ( $\text{cm}^3/\text{s K}^{3/2}$ ); R4: parameter,  $T_2$ (K); R5: parameter,  $c_3$  ( $\text{cm}^3/\text{s K}^{3/2}$ ); R6: parameter,  $T_3$ (K); R7: parameter,  $c_4$  ( $\text{cm}^3/\text{s K}^{3/2}$ ); R8: parameter,  $T_4$ (K); R9: parameter,  $c_5$  ( $\text{cm}^3/\text{s K}^{3/2}$ ); I1: nuclear charge; I2: number of electrons; I3: initial level index; I4: initial level weight; I5: xstar ion number

Mapping:

	Data/Parameter type	mapping
datatype	DRTOT_BADNELL_{A}	
parameter	PARAM_9	R9
parameter	PARAM_8	R8
parameter	PARAM_5	R5
parameter	PARAM_4	R4
parameter	ID_ION_INIT	I5
parameter	PARAM_6	R6
parameter	PARAM_1	R1
parameter	PARAM_3	R3
parameter	PARAM_2	R2
parameter	ID_{X}_INIT	I5
parameter	PARAM_14	R14
parameter	PARAM_11	R11
parameter	PARAM_10	R10
parameter	PARAM_13	R13
parameter	PARAM_12	R12
parameter	PARAM_7	R7

## A.10 Type 49

Datatype: 49

Ratetypes: 7

Description: opacity project pi x-sections

Comment: Photoionization cross section from TOPbase averaged over resonances.

Photon energies are in Ry with respect to the subshell ionization threshold and cross sections are in Mb. Just like 53, except for energy scale.. Every line contains: reals (2\*np; x(i),y(i),i=1,np) ; integers (8; N, L, 2\*J, Z, #lev+, #nion+, #nlev, #nion) ; characters (0) ; (#ion & #nlev correspond to the initial state and #ion+ & #lev+ correspond to the state to which that ionizes.

Value Description: R('1', 'N-1', '2'): energy (Ry); R('2', 'N', '2'): cross section (Mb); I1: valence n; I2: L; I3: 2J; I4: Z; I5: level index of ionized system; I6: index of ionized system; I7: level index of initial system; I8: index of initial system

Mapping:

	Data/Parameter type	mapping
datatype	PIPAR_GND_{A}{A}	R('2', 'N', '2')
parameter	ID_ION_FINAL	I6
parameter	ID_ION_INIT	I8
parameter	ENERGY_RY	R('1', 'N-1', '2')
parameter	ID_{X}_INIT	I7
parameter	ID_{X}_FINAL	I5

## A.11 Type 50

Datatype: 50

Ratetypes: 4, 9

Description: line rad. rates from OP and IP

Comment: Transition probabilities file. Every line contains: reals (3; Wavelength (A), gf-val., A-val. (cm-1)) ; integers (4; lower level, upper level, Z, #ion) ; characters (0)

Value Description: R1: wavelength (Angstroms); R2: gf-value; R3: Einstein *A* coefficient (1/s); I1: lower level index; I2: upper level index; I3: nuclear charge; I4: xstar ion number

Mapping:

	Data/Parameter type	mapping
datatype	WAVELEN_ANG_{A}{A}	R1
parameter	ID_ION_FINAL	I4
parameter	ID_ION_INIT	I4
parameter	ID_{X}_INIT	I2
parameter	ID_{X}_FINAL	I1
datatype	RADPAR_AVALUE_{A}{A}	R3
parameter	ID_ION_FINAL	I4
parameter	ID_ION_INIT	I4
parameter	ID_{X}_INIT	I2
parameter	ID_{X}_FINAL	I1
datatype	TWOPHOT_AVALUE_{A}{A}	R3
parameter	ID_ION_FINAL	I4
parameter	ID_ION_INIT	I4
parameter	ID_{X}_INIT	I2
parameter	ID_{X}_FINAL	I1

## A.12 Type 51

Datatype: 51

Ratetypes: 3

Description: iron project and chianti line collision rates

Comment: Burgess & Tully fit to collision strengths as taken from CHIANTI. Each fit entry includes the C-fitting parameter and 5 reduced collision strengths values for X=0, .25, .5, .75, 1. ; reals (7; Delta E, C fitting param., 5 Y-reduced values) ; integers (5; transition type, lower level, upper level, Z, #ion) ; characters (0)

Value Description: R1: transition energy (eV); R2: c-fitting parameter; R3: reduced collision strength at  $X = 0.00$ ; R4: reduced collision strength

at  $X = 0.25$ ; R5: reduced collision strength at  $X = 0.50$ ; R6: reduced collision strength at  $X = 0.75$ ; R7: reduced collision strength at  $X = 1.00$ ; I1: transition type; I2: lower level index; I3: upper level index; I4: nuclear charge; I5: xstar ion number

Mapping:

	Data/Parameter type	mapping
datatype	WAVELEN_EV_{A}{A}	R1
parameter	ID_ION_FINAL	I5
parameter	ID_ION_INIT	I5
parameter	ID_{X}_INIT	I3
parameter	ID_{X}_FINAL	I2
datatype	EIEPAR_CHIANTI_{A}{A}	
parameter	PARAM_5	R6
parameter	ID_{X}_FINAL	I3
parameter	ID_ION_INIT	I5
parameter	PARAM_6	R7
parameter	PARAM_1	R2
parameter	PARAM_3	R4
parameter	PARAM_2	R3
parameter	ID_ION_FINAL	I5
parameter	ID_{X}_INIT	I2
parameter	PARAM_4	R5
datatype	TRANSTYPE_{A}{A}	I1
parameter	ID_ION_FINAL	I5
parameter	ID_ION_INIT	I5
parameter	ID_{X}_INIT	I2
parameter	ID_{X}_FINAL	I3

## A.13 Type 53

Datatype: 53

Ratetypes: 1, 7

Description: opacity project pi x-sections

Comment: Photoionization cross section from TOPbase averaged over resonances.

Photon energies are in Ry with respect to the first ionization threshold and cross sections are in Mb. Every line contains: reals (2\*np; x(i), y(i), i=1,np) ; integers (8; N, L, 2\*J, Z, #lev+, #nion+, #nlev, #nion) ; characters (0) ; (#ion & #nlev correspond to the initial state and #ion+ & #lev+ correspond to the state to which that ionizes.)

Value Description: R('1', 'N-1', '2'): energy (Ry); R('2', 'N', '2'): cross section (Mb); I1: valence n; I2: L; I3: 2J; I4: Z; I5: level index of ionized system; I6: index of ionized system; I7: level index of initial system; I8: index of initial system

Mapping:

	Data/Parameter type	mapping
datatype	PIPAR_THR_{A}{A}	R('2', 'N', '2')
parameter	ID_ION_FINAL	I6
parameter	ID_ION_INIT	I8
parameter	ENERGY_RY	R('1', 'N-1', '2')
parameter	ID_{X}.INIT	I7
parameter	ID_{X}.FINAL	I5
datatype	PITOT_THR_{A}	R('2', 'N', '2')
parameter	ENERGY_RY	R('1', 'N-1', '2')
parameter	ID_ION_INIT	I8
parameter	ID_{X}.INIT	I7

## A.14 Type 54

Datatype: 54

Ratetypes: 4

Description: Transition probabilities to be computed from quantum defect or as hydrogenic.

Comment: Transition probabilities must be included as hydrogenic. reals (1; 0.0E+0) ; integers (4; lower level, upper level, Z, #ion) ; characters (0)

Value Description: R1: parameter; I1: lower level index; I2: upper level index; I3: nuclear charge; I4: xstar ion number

Mapping:

	Data/Parameter type	mapping
datatype	RADPAR_HYDFORM_{A}{A}	
parameter	ID_ION_FINAL	I4
parameter	ID_ION_INIT	I4
parameter	PARAM_1	R1
parameter	ID_{X}.INIT	I2
parameter	ID_{X}.FINAL	I1

## A.15 Type 56

Datatype: 56

Ratetypes: 3

Description: Tabulated Upsilon for HeI from Sawey & Berrington (1993).

Comment: Every line contains: reals (2n; n log10(temp), n gammas) ; integers (4; lower level, upper level, Z, #ion) ; characters (0)

**Value Description:** R('1', 'N/2', '1'):  $\log T(K)$ ; R('N/2+1', 'N', '1'): effective collision strength; I1: lower level index; I2: upper level index; I3: nuclear charge; I4: xstar ion number

**Mapping:**

	Data/Parameter type	mapping
datatype	EIEPAR_LOGT_{A}{A}	R('N/2+1', 'N', '1')
parameter	ID_ION_FINAL	I4
parameter	ID_ION_INIT	I4
parameter	TEMP_LOGK	R('1', 'N/2', '1')
parameter	ID_{X}_INIT	I1
parameter	ID_{X}_FINAL	I2

## A.16 Type 57

**Datatype:** 57

**Ratetypes:** 5

**Description:** Effective ion charge for each level to be used in collisional ionization rates

**Comment:** (same as 65) Every line contains: reals (1, Zeff) ; integers (6; N, L, 2\*J, Z, #lev, #ion) ; characters (0)

**Value Description:** R1:  $Z_{eff}$ ; I('N', 'N', '1'): xstar ion number; I('N-1', 'N-1', '1'): level index; I('N-2', 'N-2', '1'): nuclear charge; I1: valence  $n$ ; I2:  $L$ ; I3:  $2J$

**Mapping:**

	Data/Parameter type	mapping
datatype	EITOT_HYDFORM_{A}	
parameter	ID_ION_INIT	I('N', 'N', '1')
parameter	PARAM_1	R1
parameter	ID_{X}_INIT	I('N-1', 'N-1', '1')

**Note:** For ions with IC coupling there is an extra integer (3rd) for  $2J$ ; the current XML scheme cannot treat optional integers gracefully so it left out. The integer values after the optional  $2J$  (Z, level index, and ion number) are obtained from counting from the end of the list.

## A.17 Type 59

**Datatype:** 59

**Ratetypes:** 1, 7

**Description:** Total cross section for photoionizing a sub-shell using the configuration-average approximation as given by Verner & Yakovlev (A&AS 109, 125, 1995). The data is presented as a parameterized fit.

**Comment:**  $\text{sigma} = \text{sigma0} * ((y-1)^{**2} + yw^{**2}) * y^{**(-Q)} * (1+\sqrt{y/ya})^{**(-P)}$  where  $Q = 5.5 + 1 - 0.5 P$  ( $l$  is the sub-shell angular momentum quantum number) and  $y = E/E_0$ .

**Value Description:** R1: energy threshold (eV); R2:  $E_0$  (eV); R3:  $\text{sigma0}$  (Mb); R4:  $ya$ ; R5:  $P$ ; R6:  $yw$ ; I1: nuclear charge; I2: principal quantum number of ionized sub-shell; I3: angular momentum quantum number of ionized sub-shell; I4: level index of ionized system (?); I5: ion index of ionized system; I6: level index of initial system (?); I7: ion index of initial system

**Mapping:**

	Data/Parameter type	mapping
datatype	PITOT_VERYAK	
parameter	SUBSHELL	I2, I3
parameter	PARAM_5	R6
parameter	ID_CONF_INIT	I7
parameter	ID_ION_INIT	I7
parameter	PARAM_1	R2
parameter	PARAM_3	R4
parameter	PARAM_2	R3
parameter	PARAM_4	R5
parameter	ETHRESH_EV	R1
datatype	PIPAR_VERYAK	
parameter	SUBSHELL	I2, I3
parameter	PARAM_5	R6
parameter	ETHRESH_EV	R1
parameter	ID_ION_INIT	I7
parameter	PARAM_1	R2
parameter	PARAM_3	R4
parameter	PARAM_2	R3
parameter	ID_ION_FINAL	I7
parameter	PARAM_4	R5
parameter	ID_CONF_INIT	I7
parameter	ID_CONF_FINAL	I7

**Note:** The 4th and 6th integers are probably level indices, but they should always be ignored. While level data for these ions can have any coupling scheme, the parameterized expression of this data type is valid for configuration-averaged data only (we take initial and final configurations to be ground).

## A.18 Type 60

Datatype: 60

Ratetypes: 3

**Description:** Effective collision strengths fit to the form described by Callaway (ADNDT 57, 9, 1994: equation 6) which is a polynomial fit with temperature in Rydbergs. The number of parameters is variable.

**Comment:** Coefficients for analytic fits to Upsilons for H-like ions according to review by Callaway (1994; ADNDT, 57,9) Data lines contain the following information: reals (coefficients) ; integers (4; lower level, upper level, Z, ion) ; characters (5)

**Value Description:** R1: not used; R2: not used; R3: parameter, b0; R4: parameter, b1; R5: parameter, b2; R6: parameter, b3; R7: parameter, b4; R8: parameter, b5; R9: parameter, b6; I1: lower level index; I2: upper level index; I3: nuclear charge; I4: ion index

**Mapping:**

	Data/Parameter type	mapping
datatype	EIEPAR_CALLAWAY6_{A}{A}	
parameter	PARAM_5	R7
parameter	ID_{X}_FINAL	I2
parameter	ID_ION_INIT	I4
parameter	PARAM_6	R8
parameter	PARAM_1	R3
parameter	PARAM_3	R5
parameter	PARAM_2	R4
parameter	ID_ION_FINAL	I4
parameter	ID_{X}_INIT	I1
parameter	PARAM_4	R6
parameter	PARAM_7	R9

**Note:** The first two reals look to be the temperature range for validity of the fit (most likely in Ry). However, one of values for the second real casts some uncertainty on this description and so both values are left undefined.

## A.19 Type 62

Datatype: 62

Ratetypes: 3

**Description:** Effective collision strengths fit to the form described by Callaway (ADNDT 57, 9, 1994: equation 7) which is a polynomial fit with temperature in Rydbergs with an extra logarithmic/exponential term. The number of parameters is variable, but the last three non-zero parameters belong to the extra term.

**Comment:** (same as 60) Coefficients for analytic fits to Upsilons for H-like ions according to review by Callaway (1994; ADNDT, 57,9) Data lines con-

tain the following information: reals (coefficients) ; integers (4; lower level, upper level, Z, ion) ; characters (5)

**Value Description:** R1: not used; R10: parameter, b7; R11: parameter, b8; R12: parameter, b9; R2: not used; R3: parameter, b0; R4: parameter, b1; R5: parameter, b2; R6: parameter, b3; R7: parameter, b4; R8: parameter, b5; R9: parameter, b6; I1: lower level index; I2: upper level index; I3: nuclear charge; I4: ion index

**Mapping:**

	Data/Parameter type	mapping
datatype	EIEPAR_CALLAWAY7_{A}{A}	
parameter	PARAM_9	R11
parameter	PARAM_8	R10
parameter	PARAM_5	R7
parameter	ID_{X}_FINAL	I2
parameter	ID_ION_INIT	I4
parameter	PARAM_6	R8
parameter	PARAM_1	R3
parameter	PARAM_3	R5
parameter	PARAM_2	R4
parameter	ID_ION_FINAL	I4
parameter	ID_{X}_INIT	I1
parameter	PARAM_4	R6
parameter	PARAM_10	R12
parameter	PARAM_7	R9

**Note:** The first two reals look to be the temperature range for validity of the fit (most likely in Ry). However, one of values for the second real casts some uncertainty on this description and so both values are left undefined.

## A.20 Type 63

**Datatype:** 63

**Ratetypes:** 3

**Description:** h-like cij, (hlike ion)

**Comment:** Transition probabilities to be computed from quantum defect or as hydrogenic. reals (1; 0.0E+0) ; integers (4; lower level, upper level, Z, #ion) ; characters (0)

**Value Description:** I1: unknown; not used; I2: lower level index; I3: upper level index; I4: nuclear charge; I5: xstar ion number

Mapping:

	Data/Parameter type	mapping
datatype	EIEPAR_HYDFORM_{A}{A}	
parameter	ID_ION_FINAL	I5
parameter	ID_ION_INIT	I5
parameter	ID_{X}_INIT	I2
parameter	ID_{X}_FINAL	I3

## A.21 Type 67

Datatype: 67

Ratetypes: 3

Description: Fit to effective collision strengths for He-like ions by Keenan, McCann, & Kingston (Phys. Scr. 35, 432, 1987).

Comment: Every line contains: reals (3; fit coefficients) ; integers (4; lower level, upper level, Z, #ion) ; characters (0)

Value Description: R1: parameter,  $a$ ; R2: parameter,  $b$ ; R3: parameter,  $c$ ; I1: lower level index; I2: upper level index; I3: nuclear charge; I4: xstar ion number

Mapping:

	Data/Parameter type	mapping
datatype	EIEPAR_QUADSCL_{A}{A}	
parameter	ID_{X}_FINAL	I2
parameter	ID_ION_INIT	I4
parameter	PARAM_1	R1
parameter	PARAM_3	R3
parameter	PARAM_2	R2
parameter	ID_ION_FINAL	I4
parameter	ID_{X}_INIT	I1

## A.22 Type 68

Datatype: 68

Ratetypes: 3

Description: Fit to effective collision strengths for He-like ions by Zhang & Sampson.

Comment: Every line contains: reals (3; fit coefficients) ; integers (4; lower level, upper level, Z, #ion) ; characters (0)

Value Description: R1: parameter,  $a$ ; R2: parameter,  $b$ ; R3: parameter,  $c$ ; I1: lower level index; I2: upper level index; I3: nuclear charge; I4: xstar ion number

Mapping:

	Data/Parameter type	mapping
datatype	EIEPAR_QUADSCL_{A}{A}	
parameter	ID_{X}_FINAL	I2
parameter	ID_ION_INIT	I4
parameter	PARAM_1	R1
parameter	PARAM_3	R3
parameter	PARAM_2	R2
parameter	ID_ION_FINAL	I4
parameter	ID_{X}_INIT	I1

## A.23 Type 69

Datatype: 69

Ratetypes: 3

Description: Fit to effective collision strength for electron-impact excitation of He-like ions from Kato & Nakasaki (ADNDT 42, 313, 1989).

Comment: If 5 parameters, then  $\omega = A + B/X + C/X^{**2} + D/X^{**3} + E \ln X$  where  $X = E/dE$ ; if more parameters, then use previous for  $X \downarrow X_1$  and the following for  $1 \downarrow X \downarrow X_1$ :  $\omega = P*X + Q$ .

Value Description: R1: transition energy (eV); R2: parameter, A; R3: parameter, B; R4: parameter, C; R5: parameter, D; R6: parameter, E; R7: parameter, P; R8: parameter, Q; R9: parameter, X1; I1: lower level index; I2: upper level index; I3: nuclear charge; I4: xstar ion number

Mapping:

	Data/Parameter type	mapping
datatype	WAVELEN_EV_{A}{A}	R1
parameter	ID_ION_FINAL	I4
parameter	ID_ION_INIT	I4
parameter	ID_{X}_INIT	I1
parameter	ID_{X}_FINAL	I2
datatype	EIEPAR_KATNAK_{A}{A}	
parameter	PARAM_8	R9
parameter	PARAM_5	R6
parameter	ID_{X}_FINAL	I2
parameter	ID_ION_INIT	I4
parameter	PARAM_6	R7
parameter	PARAM_1	R2
parameter	PARAM_3	R4
parameter	PARAM_2	R3
parameter	ID_ION_FINAL	I4
parameter	ID_{X}_INIT	I1
parameter	PARAM_4	R5
parameter	PARAM_7	R8

## A.24 Type 70

Datatype: 70

Ratetypes: 7

Description: Coefficients for recomb. and phot x-section of superlevels.

Comment: Every line contains: reals (#; (den(i),i=1,nd),(Te(i),i=1,nt), (log10(recomb. rates(i,j),i=1,nt,j=1,nd) (ener(i), pi x-secs(i), i=1,nx) ; integers (11; nd, nt, nx, N, L, 2\*S+1, Z, #lev+, #nion+, #nlev, #nion) ; characters (0)

Value Description: R('1', 'I1', '1'): density grid ( $1/cm^3$ ); R('I1+1', 'I1+I2', '1'): temperature grid (K); R('I1+I2+1', 'I1+I2+I1\*I2', '1'): recombination rate array (1/s); R('I1+I2+I1\*I2+1', 'N-1', '2'): energy grid (Ry); R('I1+I2+I1\*I2+2', 'N', '2'): cross section array (Mb); I1: number of densities; I10: superlevel index from current ion; I11: xstar index for current ion; I2: number of temperatures; I3: number of energies; I4: n of valence sub-shell; I5: L; I6: 2S + 1; I7: Z; I8: level index from plus ion; I9: xstar index for plus ion

Mapping:

	Data/Parameter type	mapping
datatype	PIPAR.THR_{B}{A}	R('I1+I2+I1*I2+2', 'N', '2')
parameter	ID_ION_FINAL	I9
parameter	ID_ION_INIT	I11
parameter	ENERGY_RY	R('I1+I2+I1*I2+1', 'N-1', '2')
parameter	SUPER_{Y}_INIT	I10
parameter	ID_{X}_FINAL	I8
datatype	RECPAR_{A}{B}	R('I1+I2+1', 'I1+I2+I1*I2', '1')
parameter	ID_ION_INIT	I9
parameter	DENSITY_CM3	R('1', 'I1', '1')
parameter	ID_ION_FINAL	I11
parameter	ID_{X}_INIT	I8
parameter	SUPER_{Y}_FINAL	I10
parameter	TEMP_1E4K	R('I1+1', 'I1+I2', '1')

## A.25 Type 71

Datatype: 71

Ratetypes: 14

Description: Radiative transition rates from superlevels to spectroscopic levels

Comment: The data is for a grid of Ne and Te. Every line contains: reals (#; Ne(i),i=1,nd),(Te(i),i=1,nt), (rad. rates (ne,kt),kt=1,nt,ne=1, nd), Wavelength ( $\text{\AA}$ ) ; integers (6; nd, nt, lower level, upper level, Z, #ion) ; characters (0)

Value Description: R('1', 'I1', '1'): density ( $1/\text{cm}^3$ ); R('I1+1', 'I1+I2', '1'): temperature (K); R('I1+I2+1', 'I1+I2+I1\*I2', '1'): radiative rate ( $\text{cm}^3/\text{s}$ ); R('N', 'N', '1'): wavelength (Angstrom); I1: number of densities; I2: number of temperatures; I3: lower level index; I4: upper level index; I5: nuclear charge; I6: xstar ion number

Mapping:

	Data/Parameter type	mapping
datatype	WAVELEN_ANG_{B}{A}	R('N', 'N', '1')
parameter	ID_ION_FINAL	I6
parameter	ID_ION_INIT	I6
parameter	SUPER_{Y}_INIT	I4
parameter	ID_{X}_FINAL	I3
datatype	RADPAR_{B}{A}	R('I1+I2+1', 'I1+I2+I1*I2', '1')
parameter	ID_{X}_FINAL	I3
parameter	ID_ION_INIT	I6
parameter	TEMP_1E4K	R('I1+1', 'I1+I2', '1')
parameter	ID_ION_FINAL	I6
parameter	SUPER_{Y}_INIT	I4
parameter	DENSITY_CM3	R('1', 'I1', '1')

## A.26 Type 72

Datatype: 72

Ratetypes: 40

Description: Auger rates from doubly excited states.

Comment: Every line contains: reals (3; auto. rate, energy in eV above the ionization limit, statistical weight) ; integers (6; (2S+1), L, level, continuum level numb., z, ion) ; characters (10; level configuration)

Value Description: R1: Auger rate (1/s); R2: energy above IP (eV); R3: statistical weight; I1: 2S+1; I2: L; I3: initial level index; I4: continuum level index; I5: nuclear charge; I6: xstar ion number

Mapping:

	Data/Parameter type	mapping
datatype	AUTOT_{A}	R1
parameter	ID_ION_INIT	I6
parameter	ID_{X}_INIT	I3

## A.27 Type 73

Datatype: 73

Ratetypes: 3

Description: Fit to effective collision strengths from Sampson et al. for satellite levels

Comment: **of He-like ions.** Every line contains: reals (7; fit coefficients); integers (4; lower level, upper level, Z, #ion) ; characters (0)

Value Description: R1: transition energy (Ry); R2: parameter,  $z^2 S$ ; R3: parameter,  $a$ ; R4: parameter,  $c_0$ ; R5: parameter,  $c_r$ ; R6: parameter,  $c_{r+1}$ ; R7: parameter,  $r$ ; I1: lower level index; I2: upper level index; I3: nuclear charge; I4: xstar ion number

Mapping:

	Data/Parameter type	mapping
datatype	EIEPAR_SAMETAL_{A}{A}	
parameter	PARAM_5	R6
parameter	ID_{X}_FINAL	I2
parameter	ID_ION_INIT	I4
parameter	PARAM_6	R7
parameter	PARAM_1	R2
parameter	PARAM_3	R4
parameter	PARAM_2	R3
parameter	ID_ION_FINAL	I4
parameter	ID_{X}_INIT	I1
parameter	PARAM_4	R5
datatype	WAVELEN_RY_{A}{A}	R1
parameter	ID_ION_FINAL	I4
parameter	ID_ION_INIT	I4
parameter	ID_{X}_INIT	I2
parameter	ID_{X}_FINAL	I1

## A.28 Type 74

Datatype: 74

Ratetypes: 7

Description: Delta functions to add to phot. x-sections to match ADF DR recom. rates.

Comment: Every line contains: ; reals (2n+1; ionization limit (eV), (energy over g.s.(i),i=1,n), (amplitude in cm<sup>2</sup>(i),i=1,n) ; integers (8; N, L, (2S+1), Z, #lev+, #nion+, #nlev, #nion) ; characters (0)

Value Description: R('N+1)/2+1', 'N', '1'): amplitude; R('2', '(N+1)/2', '1'): energy (Ry); R1: transition energy (eV); I1: valence n; I2: L; I3: 2S + 1; I4: nuclear charge; I5: level index of plus ion; I6: ion index of plus ion; I7: level index of initial ion; I8: ion index of initial ion

Mapping:

	Data/Parameter type	mapping
datatype	PIPAR_DELTA_{A}{A}	R('N+1)/2+1', 'N', '1')
parameter	ID_ION_FINAL	I6
parameter	ID_ION_INIT	I8
parameter	ENERGY_RY	R('2', '(N+1)/2', '1')
parameter	ID_{X}.INIT	I7
parameter	ID_{X}.FINAL	I5
datatype	WAVELEN_EV_{A}{A}	R1
parameter	ID_ION_FINAL	I8
parameter	ID_ION_INIT	I6
parameter	ID_{X}.INIT	I5
parameter	ID_{X}.FINAL	I7

## A.29 Type 76

Datatype: 76

Ratetypes: 9

Description: 2 photon decay :

Comment: Just like data type 50. Every line contains: reals (3; Wavelength (A), gf-val., A-val. (cm-1)) ; integers (4; lower level, upper level, Z, #ion) ; characters (0)

Value Description: R1: Einstein A coefficient (1/s); I1: lower level index; I2: upper level index; I3: nuclear charge; I4: xstar ion number

Mapping:

	Data/Parameter type	mapping
datatype	TWOPHOT_AVALUE_{A}{A}	R1
parameter	ID_ION_FINAL	I4
parameter	ID_ION_INIT	I4
parameter	ID_{X}.INIT	I2
parameter	ID_{X}.FINAL	I1

## A.30 Type 77

Datatype: 77

Ratetypes: 23

Description: Collision transition rates from superlevels to spectroscopic levels

Comment: Every line contains: reals (#: (Ne(i),i=1,nd),(Te(i),i=1,nt), coll.rates(ne,kt),ne=1,nt,kt=1,nd), Wavelength (Å)); integers (6: nd, nt, lower level, upper level, Z, #ion); characters (0).

**Value Description:** R('1', 'I1', '1'): density ( $1/\text{cm}^3$ ); R('I1+1', 'I1+I2', '1'): temperature (K); R('I1+I2+1', 'I1+I2+I1\*I2', '1'): collision rate (1/s); R('N', 'N', '1'): wavelength (Angstroms); I1: number of densities; I2: number of temperatures; I3: lower level index; I4: upper level index; I5: nuclear charge; I6: xstar ion number

**Mapping:**

	Data/Parameter type	mapping
datatype	WAVELEN_ANG_{B}{A}	R('N', 'N', '1')
parameter	ID_ION_FINAL	I6
parameter	ID_ION_INIT	I6
parameter	SUPER_{Y}_INIT	I4
parameter	ID_{X}_FINAL	I3
datatype	EIEPAR_LOG_{A}{B}	R('I1+I2+1', 'I1+I2+I1*I2', '1')
parameter	ID_ION_INIT	I6
parameter	DENSITY_CM3	R('1', 'I1', '1')
parameter	ID_ION_FINAL	I6
parameter	ID_{X}_INIT	I3
parameter	SUPER_{Y}_FINAL	I4
parameter	TEMP_1E4K	R('I1+1', 'I1+I2', '1')

## A.31 Type 79

**Datatype:** 79

**Ratetypes:** 4

**Description:** Line data used for Auger and inner shell fluorescence calculation:

**Comment:** Same as type 4, but different data type used in order to merge with non-Auger levels when assembling database. r1=line wavelength (A) ; r2=f value ; i1=lower level index ; i2=upper level index

**Value Description:** R1: wavelength (Angstroms); R2: f-value; R3: unused; R4: unused; R5: unused; I1: lower level index; I2: upper level index; I3: xstar ion number

**Mapping:**

	Data/Parameter type	mapping
datatype	WAVELEN_ANG_{A}{A}	R1
parameter	ID_ION_FINAL	I3
parameter	ID_ION_INIT	I3
parameter	ID_{X}_INIT	I2
parameter	ID_{X}_FINAL	I3
datatype	RADPAR_FVALUE_{A}{A}	R2
parameter	ID_ION_FINAL	I3
parameter	ID_ION_INIT	I3
parameter	ID_{X}_INIT	I2
parameter	ID_{X}_FINAL	I3

Note: remove last three reals

## A.32 Type 86

Datatype: 86

Ratetypes: 41

Description: Iron K Auger data :

Comment: (from Palmer et al. 2003A&A...410..359P, Mendoza et al 2004A&A...414..377M; Palmeri et al., 2003A&A...403.1175P; Garcia et al., 2005ApJS..158...68G)  
reals r1=A<sub>ij</sub> (s<sup>-1</sup>); i1=#lev; i3=#ion)

Value Description: R1: transition energy (eV); R2: partial Auger rate (1/s); R3: total radiative rate (1/s); R4: total Auger rate (1/s); I1: level index of plus ion; I2: level index of initial ion; I3: nuclear charge; I4: ion index of plus ion; I5: ion index of initial ion

Mapping:

	Data/Parameter type	mapping
datatype	RADTOT_{A}	R3
parameter	ID_ION_INIT	I5
parameter	ID_{X}_INIT	I2
datatype	AUTOT_{A}	R4
parameter	ID_ION_INIT	I5
parameter	ID_{X}_INIT	I2
datatype	WAVELEN_EV_{A}{A}	R1
parameter	ID_ION_FINAL	I4
parameter	ID_ION_INIT	I5
parameter	ID_{X}_INIT	I2
parameter	ID_{X}_FINAL	I1
datatype	AUPAR_{A}{A}	R2
parameter	ID_ION_FINAL	I4
parameter	ID_ION_INIT	I5
parameter	ID_{X}_INIT	I2
parameter	ID_{X}_FINAL	I1

## A.33 Type 88

Datatype: 88

Ratetypes: 42

Description: Iron inner shell resonance excitation :

Comment: Photoexcitation to autoionizing levels Format is like types 49: Photon energies are in Ry with respect to the first ionization threshold and cross sections are in Mb. Every line contains: reals (2\*np;

x(i),y(i),i=1,np) ; integers (8; N, L, 2\*J, Z, #lev+, #nion+, #nlev, #nion) ; characters (0) ; (#ion & #nlev correspond to the initial state and #ion+ & #lev+ correspond to the state to which that ionize

**Value Description:** R('1', 'N-1', '2'): energy (Ry); R('2', 'N', '2'): cross section (Mb); I1: valence  $n$ ; I2:  $L$ ; I3:  $2J$ ; I4:  $Z$ ; I5: upper level index; I6: lower level index; I7: xstar ion number

**Mapping:**

	Data/Parameter type	mapping
datatype	PEPAR_RESEXC_{A}{B}	R('2', 'N', '2')
parameter	ID_ION_FINAL	I7
parameter	ID_ION_INIT	I7
parameter	SUPER_{Y}_FINAL	I5
parameter	ID_{X}_INIT	I6
parameter	ENERGY_RY	R('1', 'N-1', '2')

# Appendix B

## Database input files

All information to be inserted into the database must conform to the format described in this appendix. There are three parts to an input file. The first line of the file must start with `XSTARDB` – followed by a code indicating what kind of information the file contains. For example, the first line of an input file for a new data type must be `XSTARDB - PREP_DATATYPE`. Next comes the header, which is zero or more lines of the form `KEY: VALUE`. The header data is to hold information applying to the whole file. For example, to specify the source of a dataset, the header line is `SOURCE: NIST`, where NIST is the source. Note that header lines are recognized by a string starting at the beginning of the line containing no spaces followed immediately by a colon. The value is everything following the colon to the end of the line with leading and trailing spaces removed. Spaces (and any other character) inside the value string are permitted. The last section contains zero or more data lines. The values of each line are delimited by spaces, so no spaces are allowed as part of a value. If a space is desired, a substitute character must be used which can then be given in one of the header lines (inserting data works like this as you will see later).

The class `adminFiles` contains all the functions necessary to parse these input files (see Section 3.2.5). Each PHP script, however, expects certain header lines and data to be present. The rest of the appendix will describe each format designated by the first line code.

### B.1 PREP\_ELEMENTS

This file is used to insert elements into the database. There are no header lines and each data line contains information for a single element: nuclear charge, full name, symbol, and mass in amu.

Example: 4 Beryllium Be 9.01218.

## B.2 PREPIONS

New ions are added to the database with this file. As with the elements input file, there are no header lines. Each data line represents a single ion: ion label in astrophysical notation, ground configuration,  $2S + 1$ ,  $L$ ,  $2J$ . The configuration labels must satisfy the rules in Appendix D where the space between each sub-shell is replaced by a period. For ions with no electrons, thus no configuration, a single period needs to be used instead. Make sure to use a period in the ion string as well.

Example: He.II 1s1 2 0 1.

## B.3 PREP\_ALLLEVELS

This file is used to insert new levels into the database. The *allconfs*, *allterms*, and *alllevels* tables are modified. The input is only accepted if all levels for each configuration are present. There is no header data and each data line corresponds to a single level with elements: configuration string,  $2S + 1$ ,  $L$ , and  $2J$ . The configuration labels must satisfy the rules in Appendix D where the space between each sub-shell is replaced by a period.

Example: 2p5.3p1 3 0 2.

## B.4 PREP\_UNIT

The input file for adding new units is pretty simple. No data lines are required. A header line with key, IDENT, is needed to give the global identifier for the new unit and must be unique. A complete convention for the identifier has not yet been determined; we have been using an underscore (\_), however, to represent division (see example). There are three more header lines instructing how to display the unit in the terminal (TERM), web page (WEB), and LATEX document (LATEX).

```
XSTARDB - PREP_UNIT
IDENT: cm3_s
TERM: cm3/s
WEB: cm3/s
LATEX: cm$^3$/s
```

Note: This format will likely be expanded when the capability is added to convert quantities from one unit to another. It is not yet known how this will be implemented.

## B.5 PREP\_FUNDAMENTAL

This file is used to add a single fundamental data type. No data lines are used and there are four header lines. The unique identifier is specified with the

IDENT key. The LABEL key is used to give a more common, understandable label for the fundamental type and can include spaces (unlike the identifier). The header line with key, XID, is optional, but allows for the xstar rate type to be stored. Finally, a longer description of the fundamental type is given with the DESCRIPT key. Below is an example for partial photoionization.

```
XSTARDB - PREP_FUNDAMENTAL
IDENT: PIPAR
LABEL: partial PI
XID: 7
DESCRIPT: Partial photoionization.
```

## B.6 PREP\_PARAMTYPE

The input file for parameter types is very similar to that for adding fundamental data types except this input has no XID key. Parameters can also have units which are specified with the UNITS key; if the parameter has no unit (e.g. ion or level), then simply omit the UNITS line. Below is an example for energies in eV.

```
XSTARDB - PREP_PARAMTYPE
IDENT: ENERGY_EV
LABEL: energy (eV)
UNITS: eV
DESCRIPT: Energy in eV.
```

## B.7 PREP\_DATATYPE

The input file to add data types to the database has no data lines, but there are several header lines. Each data type needs a unique identifier which is specified with the IDENT key. A more user-friendly label is given with the LABEL key and a full description of the data type marked with the DESCRIPT key. The description should explain the full parameter set. If the data type is for a parameterized expression, try to include the formula or at least a reference where it can be found. Since the description is often long, several DESCRIPT keys can be used for the description to span several lines. There are also keys to give the unit, UNITS, and xstar data type, XID. Both of these are optional. All data types must belong to a fundamental data type; this is specified with the FUNDAMENTAL key where the value is the fundamental identifier. The PARAM key is used to give the parameter set; one PARAM line per parameter. Finally, for parameters stored in the *store2* or *store3* tables, use the DATASET key instead of the PARAM key. The value for these lines are of the form, VAR=IDENT, where VAR can be X or Y and IDENT is the parameter identifier. The fundamental data types and parameter types used in the input file must already exist in the database. See the example below for partial photoionization which uses the *store2* table.

```
XSTARDB - PREP_DATATYPE
IDENT: PIPAR_GND_ICIC
LABEL: PI, partial (ICIC)
FUNDAMENTAL: PIPAR
UNITS: Mb
XID: 49
DESCRIP: Partial photoionization cross section with respect to ionization
DESCRIP: potential of the ground state and tabulated against energy. The
DESCRIP: transition is defined as ID_ION_INIT and ID_LEVEL_INIT to
DESCRIP: ID_ION_FINAL and ID_LEVEL_FINAL.
PARAM: ID_ION_INIT
PARAM: ID_LEVEL_INIT
PARAM: ID_ION_FINAL
PARAM: ID_LEVEL_FINAL
DATASET: X=ENERGY_RY
```

## B.8 PREP\_REFERENCE

The input file is used to insert a single reference into the database. These references can then be used in sources. Each reference has a unique identifier marked by the IDENT key. For paper references, the convention used for the identifier is journal code, volume number, underscore, and page number (see example below). The full list of authors is given with the AUTHORS key. Each author name must be surname, comma, and initials. The author names are separated by semi-colons. The publication year is also specified with the YEAR key. The full reference is given with the REFERENCE key and, for papers, should have the form in the example below. Finally, web links can be given to the journal page and NASA ADS abstract page with LINK and ADSLINK.

```
XSTARDB - PREP_REFERENCE
IDENT: PRA12_2378
AUTHORS: Thomas, L D; Nesbet, R K
YEAR: 1975
REFERENCE: L D Thomas & R K Nesbet, Phys. Rev. A 12, 2378 (1975)
LINK: http://prola.aps.org/abstract/PRA/v12/i6/p2378\_1
ADSLINK: http://adsabs.harvard.edu/abs/1975PhRvA..12.2378T
```

## B.9 PREP\_SOURCE

The input file for a source has no data lines and just a few header lines. The source must have a unique identifier specified with the IDENT key. A possibly multi-line description is marked with DESC. Any notes about how the data was added to the database can be entered with the NOTES key. References can be associated with the source with the TOPREF and REF keys. Only

one TOPREF key is allowed and reserved for the most important reference. Example below.

```
XSTARDB - PREP_SOURCE
IDENT: XEIE_CH04
DESC: Electron-impact excitation data from CHIANTI: Berrington et al (1985)
DESC: and Berrington et al (1989).
NOTES: loaded from xstar type 50 data files
TOPREF: AAS125_149
REF: ADNDT33_195
REF: JPB22_665
```

## B.10 PREP DATA

Since data can come in a large number of formats, this input file can be quite complicated. Before discussing the header lines, it is beneficial to explain the data lines first. Each data line represents a single data point. All of the parameter values are listed first, then the data value, and finally optional labels which serve as parameter value substitutes. These labels are designed for use with level parameters when the levels in the data set are not well described by *LS* coupling. While each level must point to a *LS* level in the database, the actual level label from the publication can also be stored using this method.

Now on to the header lines. A source for each data set must be specified with the SOURCE key where the value is the identifier of a source already entered in the database. The data type is identified with the DATATYPE key, but the data type identifier is prefaced by a code in round parenthesis which describes the value type of the data. More on this later. There must be a PARAMETER line for each parameter in the data type definition. The order of these lines must match the order of the parameters in the data lines. As with the datatype, each parameter identifier must be preceded by a value type in round brackets. Optionally, this can also be preceded by a column label in angle brackets. The column label is used to identify a column for reference in another input file; this will be explained further in the examples. Finally, there can be LABEL keys to identify any labels following the data value. The value of each LABEL key has the same form as DATATYPE and PARAMETER where the parameter type using the label is given as the identifier.

The first example is a truncated list of level data for neutral oxygen. The data type is LEVEL\_EV\_IC and has a value type of float meaning the energy levels are floating point numbers. There are three parameter lines where the first has a label of `energy_id`, a value type of integer, and a parameter type called NULL. The NULL parameter type is a special case and not a real parameter; it allows for extra information to be in the input file that will **not** be inserted into the database. It serves as a level index here, but its real usefulness will be revealed in the next example. The next parameter type is ID\_ION with a value type of ion. The ion value type indicates that the data will be an ion string.

Value type	Description
(integer)	integer
(float)	floating point number
(string[.])	string
(element)	element label
(ion[.])	ion label
(conf[.])	configuration string
(term[.])	term string
(level[.])	level string
(filemap: <i>filename;from-&gt;to</i> )	refer to data in file with name, <i>filename</i>

Table B.1: List of input types. Some value types have square brackets; the character in the brackets is used to represent a space in the data value. Periods (.) are the default value, but can be replaced by any other character; for example [\*].

If you used the `int` value type instead, the `xstar` ion number could be used to represent the ion instead. The last parameter, with type `ID_LEVEL`, explains how the level information is stored from the data line. The value type is `level` which means that a configuration string followed by a term, underscore, and *J* value is expected. Spaces are used to separate the configuration sub-shells and the term string, but no spaces are allowed for a single data value. Therefore, a placeholder must be used. A period is used here and specified in the parameter header line inside square brackets in the value type definition. Any single-character placeholder can be used. Also note that the level parameter is also given the column label, `level_id`. Since levels can sometimes have alternate labels, we have a `LABEL` header line in this example. The value type is a string where spaces are replaced by periods. There are no alternate labels here, however, so all the label data is left blank (which becomes a single placeholder character). Now that you have had an introduction to the value types, you can see the full list of types in Table B.1.

```
XSTARDB - PREP_DATA
SOURCE: XSTAR_LEVELS
DATATYPE: (float)LEVEL_EV_IC
PARAMETER: <energy_id>(int)NULL
PARAMETER: (ion)ID_ION
PARAMETER: <level_id>(level[.])ID_LEVEL
LABEL: (string[.])ID_LEVEL
    1 01+   2p3.4S_3/2          0.00000E+00  .
    2 01+   2p3.2D_3/2          3.32518E+00  .
    3 01+   2p3.2D_5/2          3.32269E+00  .
    4 01+   2p3.2P_1/2          5.01554E+00  .
    5 01+   2p3.2P_3/2          5.01530E+00  .
```

As a second example we look at the input file for wavelengths of the same

ion as the previous example (again truncated). The purpose of this example is to explain how one input file can pull parameter value from another file; in this case level strings. As often is done, this transition data refers to a level index instead of real level labels. The database does not use level indices internally, so we must somehow convert the index to an unambiguous level label. This parameters for the initial and final levels of the transition are ID\_LEVEL\_INIT and ID\_LEVEL\_FINAL. The value types for these parameters look very complicated, but they explain how to retrieve the level label. Note that the length of these PARAMETER lines has forced us to wrap them to a new line as indicated by the backslash. In the actual file, each parameter must be fully contained on a single line and the backslashes are absent. The value type for each of these is `filemap` which takes some arguments. Immediately following the value type label is a filename which will contain the level labels. In this case, the file `pout.d030t006.level_ev_ic` is input file from our first example. After the filename is a column-to-column mapping which use the column labels in angle brackets. So, this value type is saying, "When determining these parameter values, you need to first go to the file I give you and get a mapping array from one column to another. The first column are values in the present file and the second column are values to be substituted." The rest of the file should be self-explanatory.

```
XSTARDB - PREP_DATA
SOURCE: XRAD_OX02
DATATYPE: (float)WAVELEN_ANG_ICIC
PARAMETER: (ion)ID_ION_INIT
PARAMETER: (filemap: pout.d030t006.level_ev_ic; \
            energy_id->level_id)ID_LEVEL_INIT
PARAMETER: (ion)ID_ION_FINAL
PARAMETER: (filemap: pout.d030t006.level_ev_ic; \
            energy_id->level_id)ID_LEVEL_FINAL
    01+      4      01+      3  5.02513E+07
    01+      4      01+      1  2.47097E+03
    01+      5      01+      1  2.47109E+03
    01+      5      01+      2  7.33274E+03
    01+      3      01+      1  3.72988E+03
    01+      3      01+      2  5.00690E+06
    01+      4      01+      2  7.33168E+03
    01+      2      01+      1  3.72710E+03
```



# Appendix C

## xstar Ion Numbers

Each ion in the xstar database has a unique integer assigned to it. The index is 1 for neutral hydrogen, 2 for neutral helium, 3 for singly-ionized helium, etc. Fully-stripped ions are not included in scheme. For a given nuclear charge,  $Z$ , and number of electrons,  $n_e$ , the index,  $I$ , can be calculated using

$$I = \frac{1}{2}Z(Z + 1) + 1 - n_e. \quad (\text{C.1})$$

The reverse expressions are

$$Z = \text{ceil}\left(-\frac{1}{2} + \frac{1}{2}\sqrt{8I - 3}\right) \quad (\text{C.2})$$

$$n_e = \frac{1}{2}Z(Z + 1) + 1 - I. \quad (\text{C.3})$$

where  $\text{ceil}(x)$  gives the smallest integer greater than  $x$ . In Table C.1 is a list of indices for neutral ions up to zinc ( $Z = 30$ ).

For fully stripped ions, the database scripts internally use  $-Z$  as the ion number.

Z	sym	idx	Z	sym	idx
1	H	1	16	S	121
2	He	2	17	Cl	137
3	Li	4	18	Ar	154
4	Be	7	19	K	172
5	B	11	20	Ca	191
6	C	16	21	Sc	211
7	N	22	22	Ti	232
8	O	29	23	V	254
9	F	37	24	Cr	277
10	Ne	46	25	Mn	301
11	Na	56	26	Fe	326
12	Mg	67	27	Co	352
13	Al	79	28	Ni	379
14	Si	92	29	Cu	407
15	P	106	30	Zn	436

Table C.1: xstar ion index for all neutral ions up to zinc ( $Z = 30$ ).

# Appendix D

## Configuration Strings

Configurations are stored in the database using an unambiguous notation which should be familiar to most users. A configuration consists of a space-delimited list of sub-shells in standard order each having the form,  $nlm$ , where  $nl$  is the sub-shell (standard order: 1s, 2s, 2p, 3s, ...) and  $m$  is the occupation number. Note that the shorthand notation of omitting  $m$  when unity is not used, e.g. 2s1 not 2s. Configuration strings obey the following rules:

- all closed sub-shells starting with 1s and ending just prior to the first open (or last) sub-shell are not part of the configuration string,
- the first open sub-shell is always displayed even if it is empty ( $m = 0$ ), and
- all empty sub-shells beyond the first open sub-shell are not displayed.

Some examples:

- 1s<sup>2</sup> 2s<sup>2</sup> 2p<sup>3</sup> becomes 2p3,
- 1s<sup>2</sup> 2s<sup>1</sup> 2p<sup>4</sup> becomes 2s1 2p4,
- 1s<sup>2</sup> 2s<sup>0</sup> 2p<sup>5</sup> becomes 2s0 2p5, and
- 1s<sup>1</sup> 2s<sup>2</sup> 2p<sup>4</sup> becomes 1s1 2s2 2p4.

Using a list of occupation numbers as the configuration label was considered and ultimately rejected due to the impracticality of storing Rydberg levels. Consider the configuration, 1s 200p; whereas only 13 characters are needed to store this configuration in the form described above, nearly 40 000 characters are required if using a list of occupation numbers.

To get the number of electrons of a configuration takes two steps; first you need to calculate the number of electrons in the core and then add up the occupation numbers of the visible sub-shells. To get the number of electrons in

the core,  $n_{core}$ , take the principal quantum number,  $n$ , and the orbital angular momentum,  $l$  of the first **open** sub-shell and apply the following expression:

$$n_{core} = \frac{1}{3}n(n - 1)(2n - 1) + 2l^2. \quad (\text{D.1})$$

For a configuration of 4p5 5s2 5p1 we have  $n = 4$  and  $l = 1$ . The above expression yields  $n_{core} = 30$  and the total occupation of the visible sub-shells is 8 so this configuration has 38 electrons.